Olli Pikarla

Salesforce Lightning -komponenttien suunnittelu ja toteutus

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Insinöörityö

19.4.2018

Metropolia

Tekijä Otsikko Sivumäärä Aika	Olli Pikarla Salesforce Lightning -komponenttien suunnittelu ja toteutus 42 sivua 19.4.2018
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja	Lehtori Peter Hjort

Insinöörityössä oli tarkoituksena perehtyä Salesforcen Lightning -sovelluskehyksen alkeisiin, komponenttipohjaiseen ajattelumaailmaan, komponenttien rakenteeseen sekä komponenttien kommunikointiin ja muihin Lightningin tarjoamiin ominaisuuksiin. Lisäksi tarkoituksena oli rakentaa Lightning-sovellus, jossa sovelletaan insinöörityössä esiteltyjä asioita.

Työssä rakennettiin sekä yksi komponentti, jota Salesforce ei suoraan tarjoa standardikomponenttina, että Salesforcen tietokannasta dataa hakeva ja näyttävä sovellus, jossa on reaaliaikainen nimisuodatus. Sovelluksen tarkoitus oli antaa käytännön esimerkki siitä, kuinka useat komponentit keskustelevat ja toimivat keskenään. Sekä komponentti että sovellus ja sen komponentit rakennettiin Salesforcen selainpohjaisella koodieditori Developer Consolella.

Insinöörityön lopputuloksena syntyi aloittelijan opas Salesforce-kehittäjille, jotka haluavat siirtyä kehittämään komponentteja käyttäen Lightning-sovelluskehystä.

Avainsanat	Salesforce, Lightning, komponentti, sovellus

Author	Olli Pikarla
Title	Designing and Implementing Salesforce Lightning Components
Number of Pages	42 pages
Date	19 August 2018
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation Option	Software Engineering
Instructor	Peter Hjort, Senior Lecturer

The target of this thesis was to give a guidebook for developers looking to get into Salesforce Lightning framework development and into the implementation of its component-based structure and Lightning event-based component communication. Additionally, the goal was to build a Lightning application using the knowledge gained from topics written in the thesis.

The thesis shows examples of actual component implementation using a Modal component which Salesforce doesn't offer as a standard component out of the box. The thesis also demonstrates the implementation of an Account search application that queries information from the database of Salesforce and displays it in the application. This application includes a real-time name filtering which limits the amount of displayed Accounts, and upon selecting an account, it will display even more information about the selected entry. The works in the thesis were coded using Salesforces own browser-based code editor so no external editor was needed.

Overall the thesis was successful and gives a starting point for developers and administrators looking to get into Salesforce Lightning development.

Keywords	Salesforce, Lightning, component, application

Sisällys

Lyhenteet

Apex	Force.com-alustan käyttämä ohjelmointikieli. Muistuttaa syntaksiltaan paljon Javaa.
Bootstrap	Front-end-suunnitteluun tarkoitettu tyylikomponenttikirjasto.
CORS	Cross Origin Resource Sharing. Tekniikka, joka mahdollistaa yhdeltä palvelimelta ladatun skriptin pääsyn resursseihin, jotka sijaitsevat toisen domainin palvelimella.
CRM	Customer relations management. Asiakkuuksienhallinta.
DML	Data manipulation language. Kieli, jolla voidaan lisätä, poistaa ja muokata järjestelmän tietueita, kuten tilejä- ja kontakteja.
DOM	Document Object Model. Dokumenttioliomalli. Yleinen tyyli näyttää esimerkiksi HTML tai XML dokumentin sisältö niin, että dokumentista on helppo hakea, muokata ja nähdä eri objekteja esimerkiksi JavaScriptillä.
Event	Lightning Event. Tapa kommunikoida Lightning-komponentista toiseen.
IDE	Integrated development environment. Ohjelmointiympäristö.
JSON	JavaScript Object Notation. Tapa kuvata JavaScript-objekteja normaalina yhden rivin tekstirivinä. Nimestään huolimatta se on JavaScriptistä riippumaton.
Node.js	Palvelinpään JavaScript-ajoympäristö.
SaaS	Software as a Service. Ohjelmisto, joka tarjotaan palveluna tavallisen lisenssin sijaan.
SLDS	Salesforce Lightning Design System.
SOQL	Salesforce Object Query Language. Salesforcessa käytetty DML-kieli.

SOSL	Salesforce Object Search Language. Salesforcen hakukyselyihin tarkoitettu kieli.
SVG	Scalable Vector Graphics. World Wide Web Consortiumin standardisoima vektorigrafiikan tallennusmuoto.
Visualforce	Kieli, jolla tehdään Apexiin liitettyjä sivuja. Visualforcea koodataan HTML- ja JavaScript-kielillä.

1 Johdanto

Insinöörityön tarkoituksena on auttaa ohjelmistokehittäjää ottamaan Salesforcen uusi Lightning-sovelluskehys käyttöön. Työssä tutustutaan ensin Salesforceen ja ylipäätään siihen, mitä asiakkuudenhallinta oli ja on nykyisin. Lightning-esittely aloitetaan siitä, mitä koko Lightning ylipäätään oli ja miten se eroaa vanhasta Salesforcen tekemästä Visualforce-merkkikielestä. Työn aihe on valittu siksi, että tekijä on kiinnostunut komponenttipohjaisesta ohjelmoinnista ja koska Lightning-sovelluskehys oli vielä työtä kirjoitettaessa uusi.

Työssä tutustutaan Lightning-sovelluskehyksen komponenttipohjaiseen ajattelumaailmaan ja avataan tietoa siitä, miten komponentteja tulisi ajatella, sekä perehdytään niihin osioihin, joita Lightning-komponentti sisältää ja kuinka niitä voi muokata.

Seuraavana käydään lävitse hieman Lightning-komponenttien syntaksia ja annetaan neuvoja siihen, miten komponentteja kannattaisi toteuttaa. Työssä kerrotaan myös, mitä valmiita komponentteja Salesforce tarjoaa kehittäjille, jottei kaikkia komponentteja välttämättä tarvitse alkaa koodata alusta lähtien.

Työssä tutustutaan tarkoin myös siihen, kuinka Lightning-komponenttien controller-luokat toimivat, niiden syntaksiin ja siihen, mitä keinoja Salesforce tarjoaa komponentin suorituskyvyn parantamiseen. Käydään lävitse myös tietoa siitä, kuinka dataa voidaan kuljettaa Lightning-komponentista toiseen, mikä on kehityksen kannalta välttämätöntä.

Lopuksi koodataan esimerkkikomponentti ja sovellus, joka sisältää useita komponentteja ja kommunikointia eri komponenttien välillä. Tarkoituksena on esitellä syntaksia ja sitä, kuinka Lightning-sovelluskehys toimii käytännössä.

Insinöörityöraportti on opas Lightning-sovelluskehyksen alkeisiin ja sisältää esimerkkikoodia hyödyllisten mutta helposti ymmärrettävien sovellusten rakentamiseksi.

2 Asiakkuudenhallintajärjestelmät

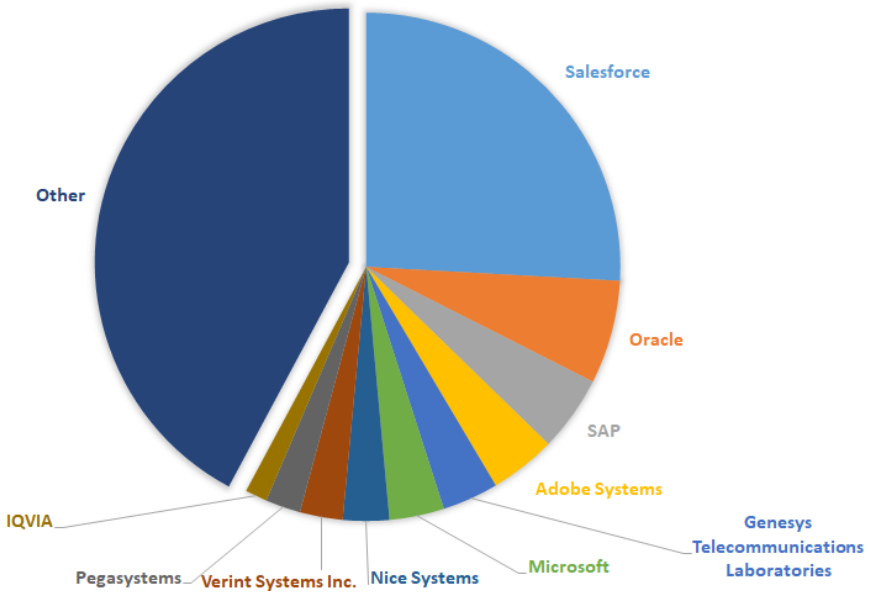
CRM (Customer relationship management) eli asiakkuudenhallinta kattaa käsitteenä käytännön asiat, strategiat ja teknologiat, joita yritykset käyttävät asiakastapahtumien analysointiin ja hallinnointiin koko asiakkuuden elinkaaren ajan. CRM-järjestelmät on suunniteltu reaaliajassa keräämään ja säilyttämään tietoa nykyisistä ja tulevista asiakkaista ja näiden käyttäytymisestä, tapahtuneista myynneistä ja kontakteista yhteen pilvipalveluun, johon monilla yrityksen henkilöillä on pääsy. Tavoitteena on parantaa liikesuhteita asiakkaiden ja yrityksen välillä, lisätä asiakasuskollisuutta ja kasvattaa myyntiä.

Asiakkuudenhallintajärjestelmät olivat aluksi kalliita ja monimutkaisia järjestelmiä, jotka asennettiin hankkivan yrityksen omiin tiloihin, mutta näiden ratkaisuiden kehittämiseen saattoi mennä kuukausista joissain tapauksissa jopa vuosiin. Näistä järjestelmistä vaihdettiin hiljalleen pilvipalveluihin, joiden ongelmana oli aluksi informaation ja automaation puute sekä ohjelmistojen heikko taso. Pilvipalveluihin haluttiin kuitenkin siirtyä, sillä ne poistivat suuret investointikulut, hankalat ohjelmistoasennukset ja päivitykset sekä muut sijoituksia vaativat esteet. Lisäksi pilvipalvelut omistavan yrityksen ei tarvinnut huolehtia jatkuvista kustannuksista, kuten ylläpitäjistä tai apuhenkilökunnan palkkaamisesta, vaan ainoaksi kustannukseksi muodostuivat itse palveluntarjoajan laskut, sillä palvelimien ylläpito jäi palveluntarjoajan hoidettavaksi.

Näitä pilvessä toimivia palveluita kutsutaan yleisesti Software as a Service -palveluiksi tai lyhyemmin SaaS-palveluiksi. SaaS-palvelut toimivat palveluntarjoajan ylläpitämällä palvelimella, jolloin ohjelmistoa ei tarvitse lainkaan asentaa, sillä tavallisimmin palvelua käytetään Internet-selaimen avulla. SaaS-palveluissa asiakas ei myöskään osta ohjelmistolisenssejä itselleen, vaan maksaa ohjelmistosta palvelumaksun yleensä kerran kuukaudessa.

CRM-SaaS-palvelut ovatkin enemmän lukittuja mutta ongelmattomampia ratkaisuja kuin alkuperäiset asiakkaiden tiloissa olevat järjestelmät olivat. Suurimpana etuna varmasti on kuitenkin se, ettei CRM-järjestelmän hankkimiseen enää tarvita suurta rahan ja ajan investointia [1].

CRM-palveluja tarjoavia yrityksiä on monia, muun muassa Microsoft, Oracle, Adobe ja Salesforce. Vuonna 2016 suurin niistä oli Salesforce (kuva 1), jota tässä projektissakin käytetään [2].



Kuva 1. CRM-toimittajien markkinaosuudet vuonna 2016 [2].

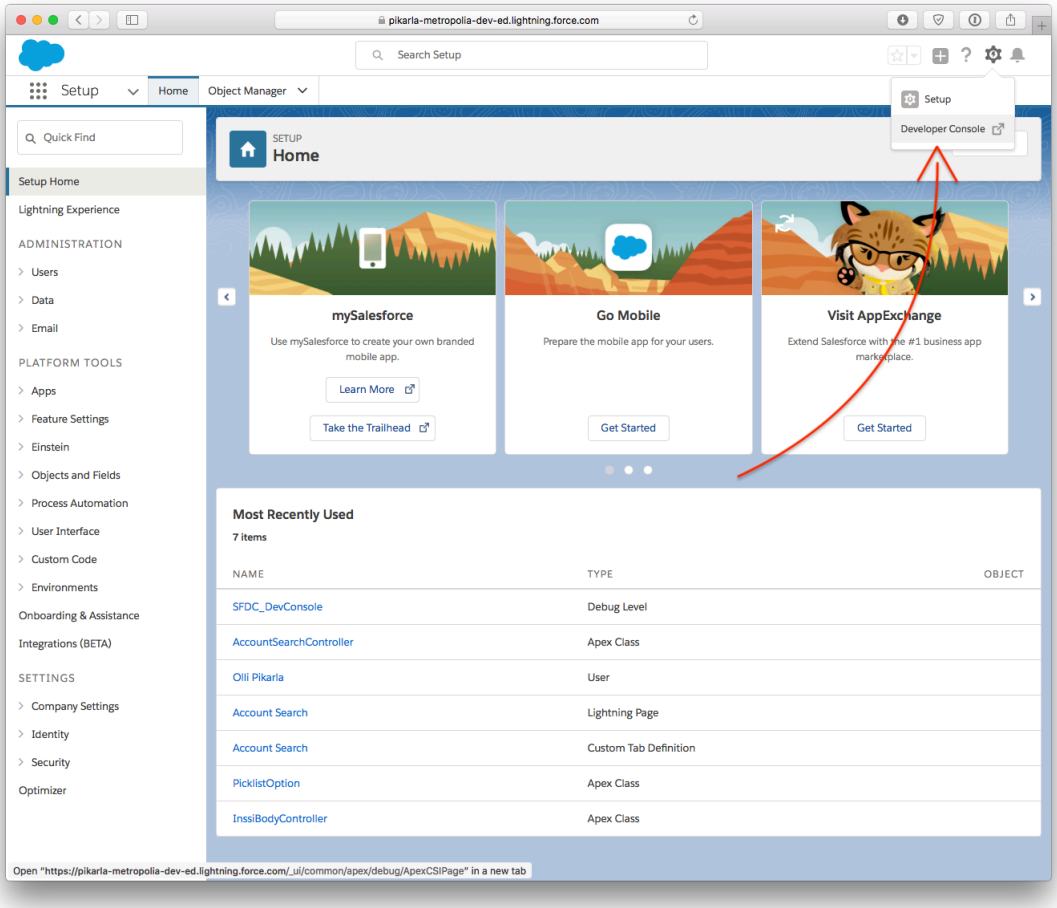
3 Salesforce ja sen osa-alueet

3.1 Yritys ja sen tuotteet

Salesforce on yhdysvaltalainen pilvipalveluita tarjoava yritys, jonka pääkonttori sijaitsee San Franciscossa. Salesforce on suurin asiakkuudenhallintajärjestelmiä tarjoava yritys, ja sen liikevaihto vuonna 2016 oli 6,667 miljardia dollaria [3]. Asiakkuudenhallintajärjestelmä on hajautettu osiin, joita ovat Analytics Cloud, App Cloud, Community Cloud, Marketing Cloud, Sales Cloud ja Service Cloud. Näistä vaihtoehdoista CRM-palvelua hankkivalle yritykselle saadaan sen tarpeisiin muokattu alusta. Alusta päivittyy kolme kertaa vuodessa ns. Spring-, Summer- ja Winter-julkaisuilla. Työtä kirjoitettaessa versio oli Spring ’18.

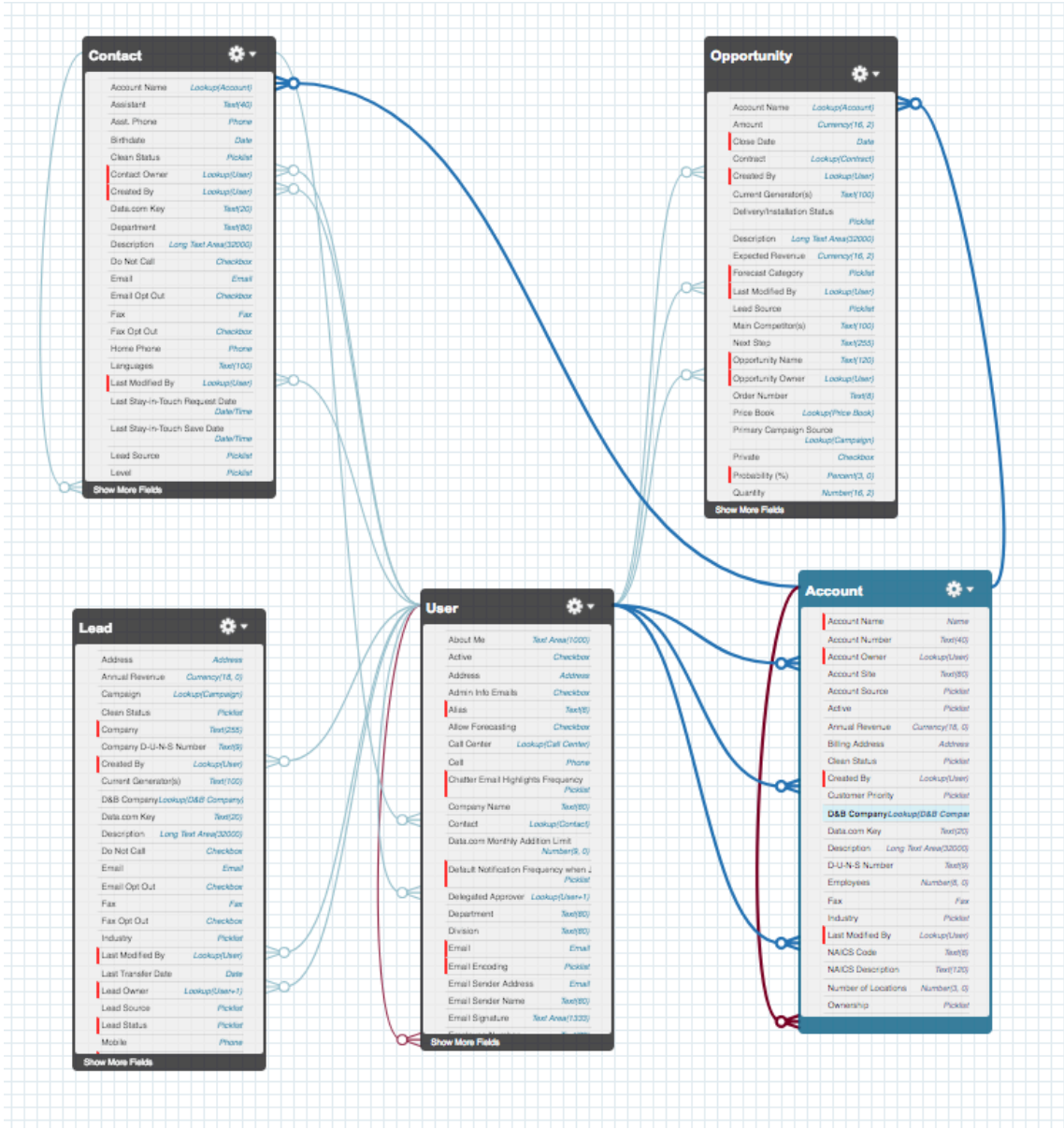
Lisäksi ylläpitäjille ja kehittäjille on olemassa Force.com-nimellä kulkeva sovelluskehitysalusta, johon kehittäjät voivat Salesforcen omalla ohjelmointikielellä koodata omia sovelluksia. Force.com toimii Salesforcen muiden palveluiden tavoin täysin pilvessä. Insinöörityön sovellus kehitettiin nimenomaan käyttäen Force.com-alustaa. Force.com-alustaa voi ohjelmoida monen eri IDE:n avustuksella mutta helpoin joskaan ei ehkä paras ratkaisu on Salesforcen oma selaimessa toimiva Developer Console (kuva 2). Developer Console toimi kuitenkin tähän työhön mainiosti ja sisältää ominaisuuksia, joita muista

editoreista ei löydy. Sillä pystyy esimerkiksi suorittamaan kyselyitä suoraan tietokantaan. Lisäksi toteutettava Lightning-sovellus päivittyy selaimessa aina itsekseen, kun koodi tallennetaan Developer Consolilla.



Kuva 2. Salesforcen Spring ’18 Lightning Experience -käyttöliittymä. Nuolesta nähdään, mistä Developer Console IDE avataan.

Salesforce sisältää oman objektipohjaisen tietorakenteen, jota pääsee hyvin katsomaan Schema Builder -ohjelmalla (kuva 3), joka on selainpohjainen ja löytyy Salesforcen Setup -valikosta. Objekteja sekä niiden kenttiä ja relaatioita voi lisätä myös Setup-valikosta. Itse tietueita voidaan lisätä sekä suoraan koodissa käyttäen Apex-luokkia että selaimessa käyttöliittymän puolella. Tässä työssä hyödynnettiin standardia Account-objektia ja muutamaa sen sisältämää kenttää.



Kuva 3. Schema Builder jossa näkyvät objektien väliset relaatiot. Kuvassa nähdään objektit Account, Contact, Lead, Opportunity ja User. Viivat kuvaavat, miten nämä objektit ovat liitoksissa toisiinsa.

3.2 Apex-ohjelmointikieli

Apex on Force.comin tekemä ohjelmointikieli, joka muistuttaa syntaksiltaan (esimerkkikoodi 1) paljon Javaa. Apexilla voidaan suorittaa erilaisia komentoja Force.com -alustalla esimerkiksi käyttäjän päivittäessä kenttiä, luodessa, muokatessa tai poistaessa tietueita, klikatessa näppäimiä tai linkkejä jne. Myös Visualforce-sivujen controller-luokat on kirjoitettu Apexilla. Kuten Java, myös Apex seuraa piste-notaatiota.

```
public static Account createTestAccount(String name)
{
      Account acc = new Account();
      acc.Name = name;
      acc.Description = 'test description here';
      return acc;
}
```

Esimerkkikoodi 1. Apex-koodia. Koodissa näkyy testidatan luomiseen käytettävä metodi.

Apexissa kaikki objektit ja kentät käyttävät päätettä ”__c” aina itsetehdyille kentille ja objekteille. Lisäksi relaatiot objektien välillä käyttävän syntaksipäätettä ”__r”

3.3 SOQL- ja SOSL-kyselykielet

SOQL on Force.comissa käytettävä SQL:n tapainen kyselykieli. Siinä missä SQL-kieltä käytetään haettaessa useista tauluista tietueita ja dataa, käytetään SOQL-kieltä haettaessa objektidataa suoraan Salesforcesta. Kyselyt voivat palauttaa dataa vain yhdestä objektista kerrallaan. Objektit voivat toki olla kytköksissä toisiinsa relaatioilla, jolloin dataa voidaan hakea ensimmäisen objektin kautta toisesta objektista. Tämän lisäksi SOQL on syntaksiltaan (esimerkkikoodi 2) pelkistetympi kuin SQL, eikä siinä ole lainkaan esimerkiksi wildcard-operaattoria tai Unique-annotaatiota kyselyn rajoittamiselle.

```
@AuraEnabled
public static List<Expense__c> getExpensesForUser(User userToSearch)
{
      return [SELECT Id, Name, Amount__c, Date__c, Owner.Name
                  FROM Expense__c
                  WHERE Owner.Id = :userToSearch.Id];
}

@AuraEnabled
public static Expense__c saveExpense(Expense__c expenseToSave)
{
      try
      {
            upsert expenseToSave;
            return expenseToSave;
      }
      catch(DmlException dmlEx)
      {
            System.debug(dmlEx);
      }

      return null;
}
```

Esimerkkikoodi 2. Haku- ja tallennusoperaatiot SOQL-kielellä. Hakuoperaatio näkyy hakasulkeissa.

Salesforce sisältää SOQL:n lisäksi toisen hakukielen, Salesforce Object Search Languagen (SOSL). Se eroaa SOQL-kielestä siinä, että se hakee useasta objektista vain yhden sijaan. Hakukysely palauttaa listan objektilistoja. Lisäksi SOSL-kieli on ainoastaan kyselyitä varten, eikä sillä voi luoda, päivittää tai poistaa dataa SOQL-kielen tapaan.

SOSL-kielellä voidaan myös suodattaa hakutuloksia pitkien tekstikenttien mukaan toisin kuin SOQL-kielessä. SOSL on hyödyllinen, mikäli kyseltävä objektin tyyppi ei ole tiedossa. SOSL-kysely on myös ainoa tapa hakea tietoa, joka on kirjoitettu kiinan, japanin, korean tai thain kielillä.

3.4 Visualforce-merkkikieli

Visualforce on Salesforcen kehittämä Force.comin käyttöliittymän sivujen kirjoittamiseen tehty merkkikieli. Visualforce mahdollistaa sivujen kehittämisen Salesforcen vakiosivujen tyyleille ja tätä kautta auttaa kehittäjien työtä, koska CSS-tyylien kanssa ei tarvitse tehtä niin paljon töitä. Visualforce myös mahdollistaa tehtyjen sivujen linkittämisen helposti suoraan Apexiin esimerkiksi tietokantakutsujen ja muun toiminnallisuuden suorittamiseen. Visualforce-sivut näyttävät koodiltaan hyvin paljon HTML5-sivuilta ja niiden tekemisessä voidaankin Apex Tagien mukana käyttää HTML-, CSS- ja JavaScript-syntaksia (esimerkkikoodi 3).

```
<apex:page controller="CustomObjPageController">

     <apex:form id="theForm">

          <apex:pageBlock title="Campsites" id="thePageBlock">

               <apex:pageBlockTable value="{!campsiteList}" var="campsite”>

                    <apex:column value="{!campsite.Id}" />
                    <apex:column value="{!campsite.Name}" />

               </apex:pageBlockTable>

          </apex:pageBlock>

    </apex:form>

</apex:page>
```

Esimerkkikoodi 3. Visualforce-sivun koodia jossa näkyy Visualforce-tageja.

3.5 Lightning-sovelluskehys

Vuonna 2015 Salesforce esitteli uuden käyttöliittymän, joka kulkee nimellä Lightning Experience. Vanha käyttöliittymä sai samalla nimen Salesforce Classic. Lightning Experience on tehty parantamaan Salesforcea käyttävän yrityksen myyjien työntekoa, joskin se sisältää suuren osan Salesforce Classic -käyttöliittymän ominaisuuksista [4].

Samaan aikaan kun Lightning Experience julkaistiin, tuli Visualforcen rinnalle uusi käyttöliittymän kehitykseen tarkoitettu sovelluskehys nimeltä Lightning. Lightning-sovelluskehys on rakennettu avoimella lähdekoodilla rakennetun Aura-sovelluskehyksen päälle. Aura on Salesforcen rakentama käyttöliittymien luontiin kehitetty avoimen lähdekoodin sovelluskehys, jolla voidaan rakentaa dynaamisia ja skaalautuvia web-sovelluksia [5]. Sillä mahdollistetaan Salesforcen palvelinpuoleen riippumattomien sovellusten teko.

Lightning perustuu komponentteihin, joita yhdistelemällä ja sisäkkäin asettelemalla voidaan rakentaa modernin näköisiä ja responsiivisia eli reaaliajassa näytön kokoon mukautuvia yksisivuisia web-sovelluksia sekä mobiili- että työpöytälaitteille. Visualforcen huonona puolena oli huono mukautuvuus mobiililaitteille sekä hitaus pienien asioiden suorittamisessa koska kaikki tapahtumat tehtiin aina palvelinpuolella.

Uusi Lightning-sovelluskehys ratkaisee nämä ongelmat. Osa datasta voidaan käsitellä valmiiksi suoraan komponenteissa itsessään, jolloin aikaa vieviä palvelinkutsuja ei tarvitsee tehdä kuin pakosta, kuten esimerkiksi datan tallennuksessa ja/tai lisädatan haussa. Komponenttipohjaisen ajattelun hyötynä kehitysaika vähenee, koska komponentteja voidaan käyttää useassa eri paikassa muutamalla koodirivin lisäyksellä. Salesforcen oma mobiilisovellus Salesforce1 on rakennettu hyödyntäen Lightning-sovelluskehystä [6].

3.6 Salesforce Lightning Design System -tyylikirjasto

Salesforce Lightning Design System (SLDS) on Salesforcen rakentama tyylien kirjoittamiseen tarkoitettu kirjasto, joka sisältyy kaikkiin uusiin Lightning-sovelluksiin, mikäli ne aloitetaan annotaatiolla ”<aura:application extends="force:slds">”. Lightning Design System toimii maailman käytetyimmän [7] tyylikirjasto Bootstrapin tavoin. Se tarjoaa

käyttäjälle suuren määrän erilaisia valmiita tyylitysluokkia esimerkiksi nappuloiden tyylitykseen, fontteihin ja fonttikokoon, asetteluun ja datan näyttämiseen.

SLDS:n ero verrattuna Bootstrapiin ja muihin tyylityskirjastoihin on siinä, että SLDS-tyylitykset ovat linjassa Salesforcen oman design-toteutuksen kanssa. Esimerkiksi kaikki nappuloiden tyylitysluokat saavat omat Lightning-komponenttinappulat näyttämään samanlaisilta kuin Salesforcen oman ympäristön nappulat. Näin saadaan rakennettua yhtenäisempää käyttäjäkokemusta ja päästään eroon siitä, että samaa asiaa tekevät palikat näyttävät joka sovelluksessa erilaisilta. SLDS-luokat ovat myös responsiivisia, joten ne mukautuvat automaattisesti, mikäli näytön koko muuttuu. SLDS tarjoaa lisäksi Salesforcen sisällä käytettäviä kuvakkeita käytettäväksi ilman erillisia resurssilatauksia Salesforce-ympäristöön. Mikäli näin halutaan kuitenkin tehdä esimerkiksi SLDS-kirjastoa muokatessa, voidaan koko kirjasto ladata ja manuaalisesti lisätä Salesforce-ympäristöön.

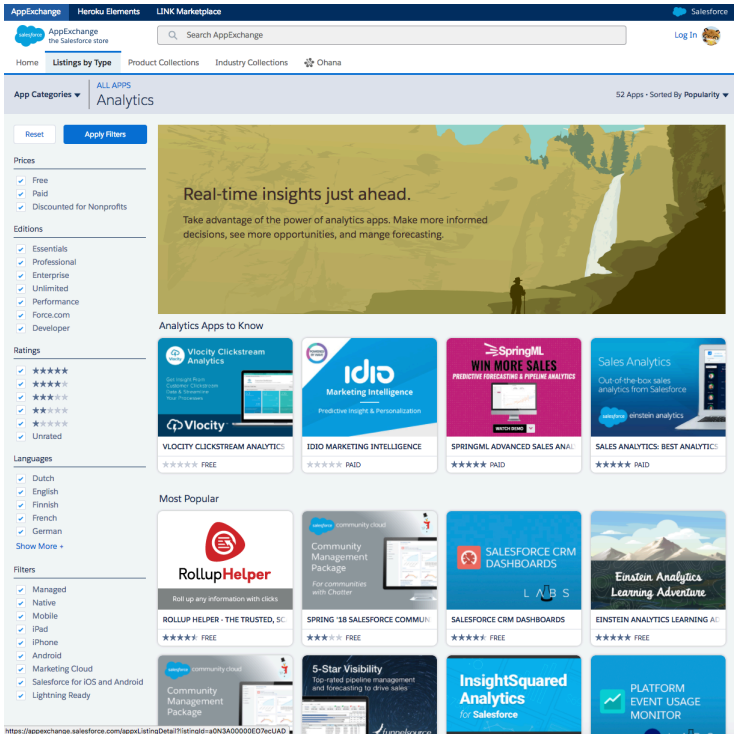
SLDS-luokkien syntaksi alkaa aina kirjaimilla ”slds”, ja se on Bootstrap-kirjaston tapaan tehty mahdollisimman helpoksi kirjoittaa. Esimerkiksi kokoa määräävä tyylityksen antaminen omalle komponentille onnistuu tyylillä ”slds-size_1-of-2”, joka rajoittaa komponentin koon puoleen tämänhetkisestä selaimen koosta.

Kaikki SLDS-kirjaston tarjoamat tyylitysluokat löytyvät osoitteesta https://www.lightning-designsystem.com/getting-started/

3.7 App Exchange -sovelluskauppa

Salesforce App Exchange on Salesforcen tarjoama sovellusten ja komponenttien kauppapaikka (kuva 4), joka toimii vähän Applen AppStoren ja Googlen Play Storen tapaan. App Exchangesta voidaan ladata Salesforce:n alustalle erilaisia isoja ja pieniä komponentteja ja kokonaisia sovelluksia. Myös Lightning-komponentteja voidaan jakaa App Exchangessa.

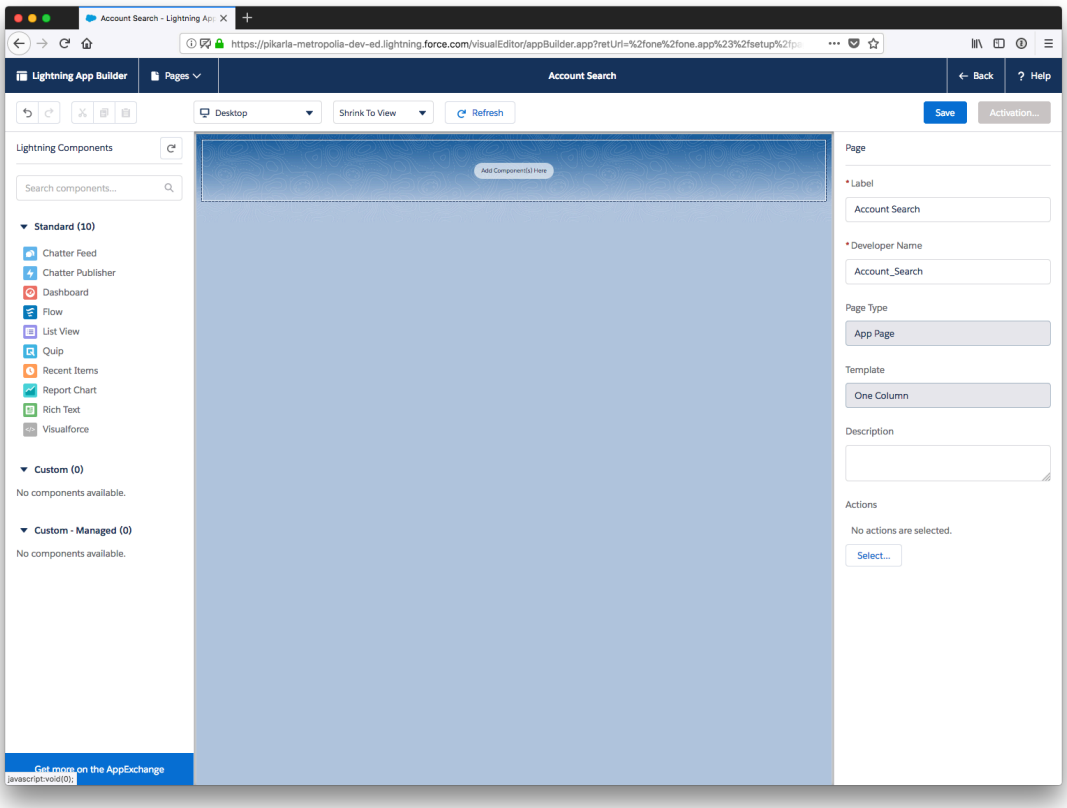
Vuonna 2016 App Exchange ylitti 3,5 miljoonan ladatun sovelluksen rajan [8]. App Exchangessa on tällä hetkellä saatavilla hieman yli 3 400 sovellusta. [9]



Kuva 4. App Exchange -verkkokauppa [7].

3.8 Lightning App Builder -rakennusalusta

Salesforcen sisällä (Setup -> User Interface -> Lightning App Builder) on Lighting App builder -sovellus (kuva 5), jonka avulla Salesforcen omaan käyttöliittymään on helppoa tehdä uusia sivuja ja sivuille upottaa itse tehtyjä komponentteja. Lighting App Builder toimii raahaa ja pudota tyylillä.



Kuva 5. Lightning App Builderin käyttöliittymä.

Sivuja, joita Lightning App Builderilla voi tehdä ovat esimerkiksi

- koko sivun sovellukset, josta nähdään standardiobjektien dataa
- Salesforcen Dashboard-raportointisivu, josta voidaan esimerkiksi seurata parhaita myyntimahdollisuuksia
- tarkkoihin yksinkertaisiin tehtäviin rakennetut sovellukset, joiden avulla myyjät voivat esimerkiksi kirjata yrityksen menoja nopeasti.

Lightning App Builder perustuu sivuihin. Sivumalleja on useita, kuten koko näytön täyttävä sivu, kahteen jaettu sivu jne. Standardikomponentit mukautuvat usein itsensä valmiiksi riippuen siitä, minkäkokoisen tilan niille antaa. Sivun luonnin jälkeen käyttöliittymän vasemmalla puolella näkyvät kaikki tarjolla olevat komponentit, joita käyttäjä voi vapaasti vetää hiirellä valitsemalleen sivutyypille.

Myös omat koodatut komponentit saadaan näkyville Lighting App Builderiin, jos oma komponentti toteuttaa ”flexipage:availableForAllPageTypes” rajapintaluokan (interface). Lisäkomponentteja saa ladattua myös App Exchangesta.

Kun halutut komponentit on siirretty tehdylle sivulle, voidaan tämä sovellus julkaista käyttäjien käytettäväksi joko selainpuolella tai Salesforcen omassa mobiilisovelluksessa Salesforce 1:ssä.

3.9 Sovellukset Salesforce-ympäristön ulkopuolella.

Salesforcen Spring ’16 julkaisussa esitelty Lightning Out -teknologia mahdollistaa valmiiden Lightning-sovellusten julkaisemisen myös itse Salesforce-alustan ulkopuolelle. Jotta tämä onnistuu, tulee Salesforce-ympäristössä olla haluttu HTML-sivu konfiguroituna Cross Origin Resource Sharing (CORS) -asetuksiin, joissa Lightning-sovellus tulee näytetyksi. Tämän lisäksi Salesforcen Communities-asetuksen on oltava päällä, sillä komponentit hakevat tiedot tätä kautta, vaikka itse Communityä ei käytettäisikään.

Halutut komponentit tulee sisällyttää Lightning-sovellukseen, johon taas sijoitetaan julkaisuun tulevat komponentit. Sivulla itsellään näytetään siten pelkästään sovellus, joka sisältää halutut komponentit. Näin Salesforcen dataa voidaan näyttää myös täysin sovelluksen ja komponentit sisältävän ympäristön ulkopuolella. Komponentit itsessään toimivat aivan suoraan eivätkä vaadi minkäänlaisia muutoksia, mutta ne sisältävä sovellus tulee luoda käyttäen esimerkkikoodin 4 syntaksia:

```
<aura:application access=”GLOBAL” extends=”ltng:outApp” implements”ltng:allow-
GuestAccess” >

     <aura:dependency resource=”c:MainComponent”/>

</aura:application>
```

Esimerkkikoodi 4. Lightning Out -sovelluksen syntaksia

Esimerkissä nähdään, kuinka Lightning Out -sovellus sisältää vain yhden Lightning-komponentin, jota käytetään paketoimaan muut mahdolliset komponentit, jotka voidaan sisällyttää kuvan MainComponent-komponenttiin. Esimerkistä saadaan myös selville, kuinka haluttu Lightning-sovellus on helppo avata muiden sivustojen käyttöön lisäämällä access- ja extends-parametreihin arvot. Mikäli sovellukseen lisätään näiden mukaan kuvassakin oleva ”ltng:allowGuestAccess”, pääsee näytettyyn dataan käsiksi myös ilman

autentikointia. Muussa tapauksessa sivulle meno aiheuttaa uudelleenohjauksen Salesforcen Login-sivulle. Autentikointi voidaan hoitaa myös itse esim. käyttäen Node.js-sovelluskehystä.

HTML-sivu jolla, komponentit on tarkoitus näyttää, vaatii kaksi eri script-tagia ja id-kohdan sivustosta siihen, mihin komponentin halutaan luonnin jälkeen sijoittaa. Esimerkissä (esimerkkikoodi 5) LightingOutTest-sovelluksen MainComponent-komponentti sijoitetaan lightningLocatoriksi nimetyn div-tagin sisään komponentin dynaamisen luontitapahtuman onnistuttua.

```
<html>
<script src=”https://community-domain/community-url/lighting/light-
ning.out.js” />
<script>
        $Lightning.use("c:LightningOutTest",
                function() {
                        $Lightning.createComponent(
                                "c:MainComponent",
                                { },
                                "lightningLocator",
                                function(cmp) {
                                    //callback funktio
                                }
                        );
                },
                //community url
                'https://community-domain/community-url'
        );
</script>
<body>
        <div id=”lightningLocator”/>
</body>
</html>
```

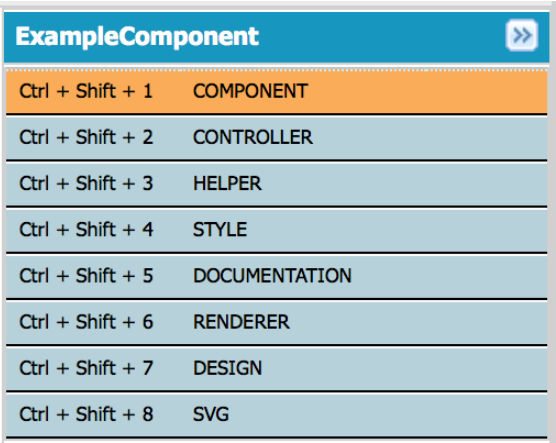
Esimerkkikoodi 5. Yksinkertainen HTML-sivu, jossa rakennetaan Lightning Out -sovellus nimeltä LightningOutTest [10].

4 Lightning-komponentit

4.1 Komponenttien rakenne

Itse Lightning-komponentteja on hyvä ajatella paketteina, jotka koostuvat yhdestä tai enintään kahdeksasta alatiedostosta (kuva 6) jotka määräävät, sisältääkö komponentti esimerkiksi kehittäjän kirjoittamia omia tyyliluokkia tai dokumentointia. Komponenttia

luodessa päätetään, mitkä tiedostot komponentti sisältää ja tarvittaessa tiedostoja on helppo lisätä mukaan.



Kuva 6. Salesforcen Developer-konsolissa avattu komponentti. Kuvassa näkyvät komponentin eri osat.

Component

Ensimmäinen tiedosto sisältää komponentin rakenteen eli koodin siitä, mitä näytetään ja missäkin kohdassa. Komponentti koodataan HTML5-kielellä, ja se voi sisältää kaikkia normaaleja HTML5-tageja, kuten ” ”, ”<h1>”, ”<h2>” jne. Näiden lisäksi se voi sisältää myös Lightningille tehtyjä omia tageja, jotka aloitetaan annotaatiolla ”aura”.

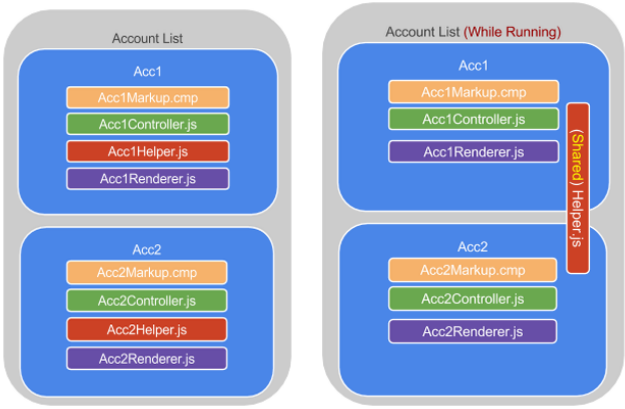
Controller

Toinen tiedosto sisältää komponentin controller-luokan. Controller sisältää komponentin logiikkan, kuten tyylitysluokkien lisäyksen ja poiston komponenttia näytettäessä, komponenttiin lisätyn datan hakemisen käsiteltäväksi, uuden datan laittamisen esiteltäväksi ja Apex-koodin kutsut esimerkiksi DML (Data Manipulation Language) -operaatioiden suorittamiseksi.

Helper

Kolmas tiedosto on niin kutsuttu helper-tiedosto, joka sisältää controller-luokan tapaan toiminnallisuutta. Helper-luokan metodeja ei voi kutsua suoraan komponentista, vaan niitä on kutsuttava itse controller-luokasta piste-notaatiolla. Helper- ja controller-luokkien

ero on siinä, että komponentteja luotaessa saa jokainen yksittäinen komponentti oman controller-luokkansa, kun taas helper-luokkia tehdään kaikille samannimisille komponenteille vain yksi (kuva 7).



Kuva 7. Helper-luokan selitys. Vasemmalla nähdään, miltä kaksi samanlaista komponenttia näyttää visuaalisesti, kun taas oikealla näkyy, kuinka järjestelmä ymmärtää ne todellisuudessa.

Style

Neljäs tiedosto sisältää komponentin rakenteen käyttämät tyylitykset, jotka koodataan CSS-kielellä. CSS-luokkien syntaksin alkuosa tulee aloittaa joko annotaatiolla .THIS ja luokan nimi tai jättämällä välilyönti .THIS ja luokan-nimen jälkeen pois (esimerkkikoodi 6).

```
.THIS.centerText
{
      text-align: center;
}
.THIS .boldFont
{
      font-weight: bold;
}
```

Esimerkkikoodi 6. Lightning Style class -syntaksia.

Näiden kahden syntaksin erona on se, että tyylityksissä joissa ”.THIS”-annotaatio ja luokan-nimi on kirjoitettu yhteen, käytetään tyyliä vain ylimmän tason tageissa, joissa viitataan kyseessä olevaan luokkaan. Mikäli välilyönti lisätään, käytetään tyyliä kaikkiin mahdollisiin tageihin, joissa tyylityksiin viitataan [11].

Documentation

Viides tiedosto on dokumentaatiota varten. Koska komponentit ovat uudelleen käytettäviä, on monimutkaisimpiin komponentteihin hyvä olla dokumentaatiota. Se voidaan kirjoittaa suoraan komponenttiin itseensä. Dokumentaatio tukee tavallisen HTML5-koodin lisäksi myös Aura-komponentteja ”description” ja ”example”. Description-komponentin sisään kirjoitetaan itse dokumentaatio kyseessä olevasta komponentista. Example-komponentin sisään voidaan liittää viitteitä siihen, miltä komponentti näyttää eri tilanteissa.

Kaikissa Salesforce-ympäristöissä on sisäänrakennettu referenssisovellus, jossa listataan kaikki luokat, tapahtumat ja komponentit. Dokumentaatio-osuuteen kirjoitetut ohjeet näkyvät tässä referenssisovelluksessa. Referenssisovellus on tätä kirjoitettaessa vasta Beta-vaiheessa, ja siihen pääsee käsiksi vain manuaalisesti kirjoittamalla selaimen osoitepalkkiin oman Salesforce-ympäristön osoitteen ja merkitsemällä sen perään ”/componentReference/suite.app”.

Renderer ja luonnin aikajana

Kuudes tiedosto on niin kutsuttu Renderer, jonka avulla voidaan ylikirjoittaa komponentin piirtovaiheessa tapahtuvaa logiikkaa. Komponentin luonnin aikajana menee seuravasti:

1. Sovelluskehys kutsuu mahdollista alustusmetodia, jolloin voidaan alustaa dataa tai attribuutteja tai vaikka laukaista tapahtumia.

2. Seuraavaksi sovelluskehys kutsuu komponentin render-metodia, jolloin itse komponentin osat luodaan. Render-metodin jälkeen DOMin eli sovelluksen tai komponentin dokumenttiobjektimallin luonti on valmis.

3. Seuraavaksi kutsutaan afterRender-metodia, jolloin on ensimmäinen hetki päästä muokkaamaan jo piirrettyjä komponentteja.

Komponentin piirtäminen on jaettu neljään eri metodiin. Mikäli niitä halutaan muokata, tulee oman metodin olla nimetty samalla nimellä kuin jokin näistä neljästä. Muussa tapauksessa metodia ei kutsuta.

Ensimmäinen metodeista on render-metodi (kuva 8), jota kutsutaan komponenttia luodessa. Se suoritetaan ennen kuin mitään on piirretty näytölle, mutta kuitenkin mahdollisen alustusmetodin jälkeen. Render-metodi eroaa muista siinä, että sen on palautettava arvoja. Muokattavaksi arvoksi saadaan alkuperäisen render-metodin arvot käyttämällä ”this.superRender()”-metodia, joka palauttaa komponentin DOMin, jota pääsee sen jälkeen muokkaamaan, ennen kuin mitään piirretään näytölle.

```
render : function(component, helper)
{
     var dom = this.superRender();
     //Oma render-metodin koodi kirjoitetaan tähän
     return dom;
}
```

Esimerkkikoodi 7. Ylikirjoitettu render-metodi. Kaikista renderer-luokan metodeista vain render-metodin tulee palauttaa arvo. Syntaksi ”this.superRender()” palauttaa modostettavan DOMin ennen kuin mitään piirretään ruudulle.

Toinen metodi on ”afterRender()”, jota kutsutaan heti ensimmäisen render-metodikutsun jälkeen. Erona ensimmäiseen render-metodiin verrattuna on se, että kaikki DOMin objektit ovat nyt luotu näytölle. Tämä metodi ei palauta arvoa. Yleensä on hyvä lisätä logiikka jo valmiina olevalle afterRender-metodille, joten sen ensimmäisenä rivinä on hyvä olla rivi ”this.superAfterRender()”, joka ajaa standardimetodin ensin.

Kolmantena on ”rerender()”-metodi, jota Lightning-sovelluskehys kutsuu aina, kun komponentissa muuttuu jokin arvo esimerkiksi käsitellyn Lightning-tapahtuman tai nappulan painalluksen vuoksi. Kuten edellä oleva afterRender, ei reRender-metodikaan palauta arvoa. Kuten afterRender-metodissa, on myös tässä metodissa hyvä kutsua standardia toiminnallisuutta ennen omia muutoksia, jotta koko funktion ydintoiminnallisuus ei hajoa. Tämä kutsu tapahtuu syntaksilla ”this.superRerenderer();”

Neljäs ja viimeinen metodi on ”unrender()”, jota kutsutaan, mikäli komponentti tuhotaan esimerkiksi controllerissa. Tämäkään metodi ei palauta arvoa, mutta kuten edellä olevissa metodeissa, myös unrender-metodissa on hyvä kutsua ensin standardia funktiota koodilla ”this.superUnrenderer();”.

Komponentin render-tiedosto on alkuvaiheessa tyhjä, mutta nämä neljä metodia kuitenkin löytyvät taustalta, ja ne voidaan ylikirjoittaa, kunhan metodin nimi on jokin edellä mainitusta neljästä. Käytännössä en ole vielä kohdannut paikkaa, jossa olisin joutunut ylikirjoittamaan näitä metodeja uusiksi, sillä yleensä alustus metodit riittävät mainiosti, koska

data halutaan saada komponentin käytettäväksi mahdollisimman aikaisessa vaiheessa ja alustusmetodit suoritetaan ennen render-metodeja.

Design

Design-resursseilla voidaan käyttäjille antaa mahdollisuus asettaa attribute-komponenttien arvoja, kun komponentteja ei luoda koodissa vaan esimerkiksi Lightning App Builder -sovelluksessa tai Community Builder -sovelluksessa. Esimerkiksi komponentti, joka näyttää otsikkotekstin, voidaan tehdä niin, että App Builderissä komponenttia lisättäessä päästään määrittämään näytettävä otsikko koskematta koodiin. Halutut muuttujat julkaistaan komponentin ulkopuolelle käyttäen syntaksia ”<design:attribute name="componentAttributeName" label=”Nimike”/>”, jossa componentAttributeName on komponenttipuolen attribuutti, johon design-attribuuttiin annettu arvo sijoitetaan, ja label on haluttu nimike, joka käyttäjälle näkyy tietoa syötettäessä.

SVG

Viimeisenä komponenttirakenteessa on SVG-resurssi. SVG-resurssin avulla voidaan määrätä Lightning-komponentin kuvake Lightning App Builderissä ja Community Builderissä. Salesforce suosittelee SVG-resurssin enimmäiskooksi 150 kB ja sen tulisi olla enimmillään 160 pikseliä korkea ja 560 pikseliä leveä.

4.2 Komponenttien luonti

Kaikki komponentit kuuluvat nimiavaruuteen, jonka avulla komponentit on helppo ryhmitellä. Mikäli Salesforce-ympäristöön on asetettu nimiavaruuden alkuliite, voidaan omiin Lightning-komponentteihin viitata syntaksilla ”<namespace:componentName/>”, jossa namespace on ympäristön nimiavaruuden alkuliite ja componentName on halutun komponentin nimi. Mikäli alkuliitettä ei ole, viitataan komponentteihin syntaksilla ”<c:componentName/>”.

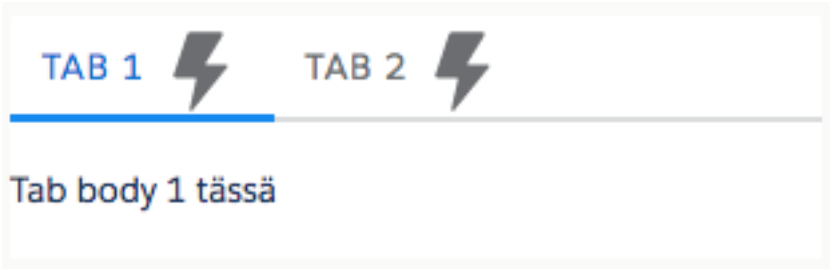
Jotta kaikkea ei tarvitse alkaa koodata tyhjästä, voidaan omissa komponenteissa käyttää Salesforcen valmiiksi rakennettuja yleisiä komponentteja. Nämä komponentit löytyvät nimiavaruuksien aura, force, lightning (kuva 8), ui ja waven alta. Tätä työtä tehdessä

valmiita komponentteja on saatavilla 143 ja osa niistä on vasta beta-versioina [12]. Uusia komponentteja tulee jatkuvasti lisää ja vanhoja päivitetään.

```
<lightning:tabset>
     <lightning:tab>
          <aura:set attribute=”label”>
               Tab 1
               <lightning:icon iconName=”utility:connected_apps”/>
          </aura:set>
          Tab body 1 tässä
     </lightning:tab>
     <lightning:tab>
          <aura:set attribute=”label”>
               Tab 2
               <lightning:icon iconName=”utility:connected_apps”/>
          </aura:set>
          Tab body 2 tässä
     </lightning:tab>
</lightning:tabset>
```

Esimerkkikoodi 8. Standardi komponentti välilehtien tekoon. Komponentissa käytetään valmiita Lightning-nimiavaruuden komponentteja.

Kun komponentti ladataan, piirtyy esimerkkikoodin komponentti näytölle välilehtinä (kuva 8).



Kuva 8. Lightning tabset-ja tab-komponentit.

Suurin osa näistä yleisistä komponenteista on hyvin alhaisella tasolla olevia komponentteja esimerkiksi tekstin syöttöön, kuvan näyttämiseen, painettaviin nappuloihin, latausindikaattorin näyttämiseen, tiedostojen lataamiseen yms. Valmiiden komponenttien hyvänä puolena on aina ajantasainen koodi ja se, että nämä komponentit näyttävät samalta kuin Salesforcen omat käyttöliittymän komponentit.

Dynaaminen luonti

Lightning-controllereissa on mahdollista luoda komponentteja dynaamisesti koodissa. Syntaksi komponentin luonnille on ”$A.createComponent(String type, Object attributes, function callback)”, jossa ”type” on luotavan komponentin tyyppi (esimerkiksi ”lighting:button”), ”attributes” on objekti niistä attribuuteista joita komponentti ottaa, kuten nappulan tapauksessa esimerkiksi ”onclick”-funktion tiedot, ja viimeiseksi callback-funktio, joka suoritetaan komponentin luonnin jälkeen. Callback-funktiossa luotu komponentti asetetaan luovan komponentin body-tagin sisään. Esimerkissä on dynaaminen luomisprosessi, jossa komponentin (esimerkkikoodi 9) sisälle luodaan uusi komponentti (esimerkkikoodi 10).

```
<aura:handler name=”init” value=”{!this}” action=”{!c.doInit}”/>

<p>
     Dynaamisesti luotu komponentti sijoitetaan komponentin rakenteeseen
     {!v.body}
</p>
```

Esimerkkikoodi 9. Komponentti, johon uusi dynaamisesti luotava komponentti sijoitetaan “{!v.body}”-tagin kohdalle.

On huomioitava että luovan komponentin rakenteessa on oltava spesifioitu ”{!v.body}” syntaksilla se, missä komponentin rakenne sijaitsee, mikäli siihen haluaa controllerissa kajota.

```
doInit : function(component)
{
     $A.createComponent(
          ”lightning:button”,                          //rakennettava komponentti
          {                                            //komponentin attribuutit
               ”aura:Id”: ”newButtonId”,
               ”label”: ”Click here”,
               ”onclick”: component.getReference(”c.buttonClicked”)
          },
          function(newButton, status, errorMessage) // Callback-funktio jossa
                                                       uusi komponentti laitetaan
                                                       näkyville
          {
               if(status === “SUCCESS”)
               {
                    var body = component.get(“v.body”);
                    body.push(newButton);
                    component.set(“v.body”, body”);
               }
          }
     );
},
buttonClicked : function(component, event, helper){},
```

Esimerkkikoodi 10. Dynaaminen komponentin luontiprosessi.

4.3 Aura-komponentit

Koska Lightning perustuu Salesforcen rakentamaan avoimen lähdekoodin Aura-sovelluskehykseen, voidaan itsetehdyissä komponenteissa käyttää lisäksi Aura-komponentteja hyödyksi. Nämä Aura-komponentit toimivat logiikkametodeina komponenttien struktuurissa. Olen käytännössä huomannut, että selvästi käytetyimmät Aura-komponentit ovat attribute (esimerkkikoodi 11), iteration (esimerkkikoodi 11), if (esimerkkikoodi 12), registerEvent ja handler (esimerkkikoodi 13).

Komponentissa mahdollisesti käytettävät muuttujan arvot ja komponentin käsiteltävä data on tallennettava attribute-komponentin sisään. Näitä attribute-komponenttien asetteluja voidaan tehdä komponentin controller- tai helper-tiedostossa, tai ne voidaan periä ylemmän tason komponentilta sitä luotaessa.

```
<aura:attribute name=”accounts” type=”Account[]” default=”[]”/>

<aura:iteration items”{!v.accounts}” var”account”>

     {!account.Name}

</aura:iteration>
```

Esimerkkikoodi 11. Esimerkkisyntaksia attribute- ja iteration-komponenteista.

Komponenteissa kuten muuallakin koodimaailmassa ehtolauseena toimii ”if”, jolle annetaan luontivaiheessa mahdollinen ”else”-lauseke ”set”-komponenttina (esimerkkikoodi 12).

```
<aura:attribute name=”exampleBoolean” default=”true”/>

<aura:if isTrue=”{!v.exampleBoolean”}>
          True
     <aura:set attribute=”else”>
          False
     </aura:set>
</aura:if>
```

Esimerkkikoodi 12. If-komponentin syntaksi.

Handler-komponenttia käytetään sekä tapahtumien suorittamisessa että komponentin luontivaiheessa mahdollisen alustusmetodin kutsumisessa. Näiden lisäksi, handler-

komponentti voidaan laittaa odottamaan muutosta tiettyyn attribuuttiin ja kutsumaan controller-luokan metodia muuttujan arvon muuttuessa.

```
<aura:handler name="init" value="{!this}" action="{!c.doInit}"/>

<aura:handler event="c:exampleEvent" action="{!c.example}"/>

<aura:handler name="change" value="{!v.attributeName}" action="{!c.example}"/>
```

Esimerkkikoodi 13. Handler-komponentteja. Ensimmäinen kutsuu metodia komponentin luomisprosessin alussa. Toinen kutsuu metodia Lightning-tapahtuman saapuessa. Kolmas kutsuu controller-metodia komponentin attribuutin arvon muuttuessa.

4.4 Viittaukset

Muuttujiin ja metodeihin viittauksen komponenttipuolella kirjoitetaan hakasulkeisiin, ja ne alkavat aina syntaksilla ”{!”, jota seuraa joko ”v.”, kun viittauksen kohteena on komponentin attribuutti, tai ”c.”, jos kyseessä on controller-luokan metodi, esimerkiksi ”v.attributeName”, jossa attributeName on halutun attribuutin nimi tai c.methodName”, jossa methodName on halutun metodin nimi. Nämä viittaukset toimivat toimivat sekä itse komponentissa että komponentin controller- ja helper-luokissa.

On huomioitava, että controller-luokasta ei voi kutsua muita controller-luokan metodeja, mutta helper-luokan metodeja kuitenkin on mahdollista kutsua. Tämä tapahtuu etuliiteellä ”helper.methodName”, jossa methodName on helper-luokassa esiintyvän metodin nimi.

Jos halutaan kutsua Salesforcen taka-alalla toimivaa Apex-controller-luokkaa, on käytössä sama syntaksi kuin controlleria kutsutessa komponentin puolelta, mutta Apex-controlleria ei voi kutsua suoraan komponentista itsestään vaan ainoastaan sen controller- tai helper-luokista.

4.5 Controller-luokat

Controllerien koodi kirjoitetaan käyttäen JavaScript-ohjelmointikieltä. Paitsi Aura-komponenttien logiikkapaloja, kaikki komponenttien logiikka tehdään komponentin controllereissa. Kaikki tyyliluokkien lisäykset ja muutokset, attribuuttien muutokset, datan

muutokset jne. on tehtävä joko itse komponentin controller- tai helper-luokassa. Lisäksi kaikki komponentin käyttämä data on joko haettava controllerissa palvelinpuolelta tai se on annettava attribuuttina komponentille suoraan sitä luodessa.

Sekä controller että helper-luokkien metodit tulee kirjoittaa esimerkin (esimerkkikoodi 14) mukaisella syntaksilla, ja eri metodit tulee erotella toisitaan pilkulla. Metodeille voi antaa argumentteja, kuten tavallisille metodeille JavaScript-koodissa.

Nyrkkisääntönä on hyvä antaa aina kolme perusargumenttia sekä controller- että helper-luokkien metodeille. Nämä argumentit ovat ”component, event, helper”. Nämä kolme argumenttiä takaavat, että kyseessä oleva metodi voi käyttää komponentin attribuuttjea, kysellä mahdollisen Apex-controller-luokan metodeja, käsitellä komponentilta saapuvia tapahtumia kuten näppäinpainalluksia sekä kutsua komponentin helper-luokan metodeja.

```
({
      openModal : function(component, event, helper)
      {
            component.set('v.showModal', true);
      },

      closeModal : function(component, event, helper)
      {
            component.set('v.showModal', false);
      },
})
```

Esimerkkikoodi 14. Komponentin controller-luokan metodeja.

Alustusmetodi

Usein eteen tulee tilanne, jossa dataa on haettava attribuutteihin jo ennen komponentin näyttämistä, sillä esimerkiksi kontakteja listaava sovellus näyttäisi muuten ensin tyhjältä. Tämä onnistuu käyttämällä Init-tapahtumaa. Init on Salesforcen valmiiksi luoma tapahtuma, joka on automaattisesti kaikkien komponenttien käytössä. Tämä tapahtuma eroaa muista siinä, että se suoritetaan kokonaisuudessaan heti komponentin alustuksen jälkeen ja ennen kuin komponentti piirretään näytölle. Init-tapahtumaa kutsutaan rekisteröimällä komponentille handler-komponentti (esimerkkikoodi 15). Se rekisteröi komponentin kutsumaan sen controllerissa olevaa ”doInit”-metodia.

```
<aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
```

Esimerkkikoodi 15. Init-tapahtuman kutsuminen komponentissa.

Konventiona Lightningissä on nimetä tämä aloitusmetodi nimellä ”doInit”, mutta tämä ei ole pakollista koodin toimivuuden vuoksi, kunhan yllä mainitussa handler-komponentissa kutsutaan oikeaa metodia. Itse logiikka täytyy kirjoittaa ”doInit”-metodin sisään. Oma kokemukseni on, että suurin osa Init-tapahtuman teoista on alkudatan hakua ja sen tarjoamista komponenttien attribuutteihin käytettäväksi.

Attribuuttien ja komponenttien hakeminen ja asetus

Controller- ja helper-luokat ovat ainoat paikat joissa attribuutteja voidaan muokata ja asettaa. Attribuuttien haku tapahtuu käyttämällä metodeissa syntaksia ”component.get(’v.attribuutti’)”, jossa ”component” on ensimmäinen metodille annettu argumentti, ja ”attribuutti” on komponentilta haettavan attribuutin nimi. Tämä syntaksi palauttaa haettavan attribuutin, mikäli se löytyy.

Jos halutaan etsiä komponentin sisällä olevia toisia komponentteja, voidaan käyttää syntaksia ”component.find(’v.komponentinNimi’)”, jossa ”component” on jälleen ensimmäinen metodille annettu argumentti ja ”komponentinNimi” vastaa halutulle komponentille annettua aura:id-arvoa. Tämä metodi palauttaa halutun komponentin tai taulukon komponentteja, mikäli useampi aura:id-tagin arvo on löydetty. Palautettuja komponentteja voidaan muokata esimerkiksi tyylitiedostoja lisäämällä tai poistamalla mutta niiden attribuutteihin ei päästä käsiksi.

Komponentin attribuuttien asetus voidaan toteuttaa syntaksilla ”component.set(’v.attribuutinNimi’, ’uusiArvo’)”, jossa ”component” on jälleen ensimmäinen metodille annettu argumentti, ”attribuutinNimi” on asetettavan attribuutin nimi ja ”uusiArvo” on attribuutille asetettava arvo. Attribuuttien asettelussa on oltava tarkkana, sillä ohjelma usein kaatuu, mikäli arvoa yritetään asettaa attribuutille, jota ei ole olemassa.

Apex controllerin kutsuminen

Kunhan komponentille on asetettu Apex-controller-luokka, voidaan tämän luokan metodeja kutsua Lightning-controllerista.

Apex-metodikutsu rakennetaan komponentin controllerissa JavaScript-muuttujaksi, ja sille annetaan mahdolliset parametrit, jotka toimivat Apex-metodin mahdollisina parametreinä. Parametrien nimien on vastattava Apex-metodin parametrien nimiä.

Seuraavaksi metodikutsulle määrätään ns. Callback-funktio. Callbackillä tarkoitetaan tapahtumaa, joka tehdään Apex-kutsun tultua suoritetuksi. Mikäli esimerkiksi Apexilta odotetaan palautuvan listan objekteja, täytyy Callback-funktiossa asettaa palautettu arvo haluttuun attribuuttiin, jotta komponentti pääsee käsittelemään palautunutta dataa.

Ennen Apexin palautettavien arvojen asettelua kannattaa varmistaa vielä, että palautettavan arvon tila on onnistunut ja että komponenttia ei ole ennen palvelinkutsunkutsun palautumista ehditty tuhota. Nämä ovat ennaltaehkäiseviä toimia sitä varten, ettei sovellus kaadu esimerkiksi palautuneen arvon korruptoitumiseen, Apex-metodin suorittamisen epäonnistumiseen tai siihen, että palautettua arvoa yritetään laittaa attribuuttiin, joka ei ole olemassa.

Apex-kutsu pitää vielä lopuksi lähettää käyttäen syntaksia ”$A.enqueueAction(kutsu)”, jossa ”kutsu” on rakennetun Apex-kutsun nimi. ”$A”-syntaksi tarkoittaa globaalia instanssin nimeä.

Insinöörityötä tehtäessä oli tiedon palauttamisessa Salesforcen palvelimesta Lightning-komponenteille vielä puutteita. Silloin tällöin varsinkin itse lisättyjä objekteja palautettaessa listana tai objektina, näytti kyseinen lista tai objekti Apex-luokassa hyvältä mutta saattoi saapua Lightning puolelle palautuessa täysin tyhjänä tai ”null”-arvolla. Tämä voidaan kiertää palauttamalla monimutkaisemmat objektirakenteet Apex-puolelta käyttäen JSON-tekstiä. Sekä Apex, että nykyaikaiset selaimet tukevat JSON-objektien ja tekstin muodostamista ja purkua, joten hyvänä vinkkinä virheiden välttämiseksi on tehdä Apexin ja Lightning-controllerien välinen datanvaihtaminen JSON-tekstinä. Lightning-komponentin päässä voidaan JavaScript-muuttujasta tehdä JSON-teksti syntaksilla ”JSON.stringify(muuttuja)”, jossa muuttuja on halutun JavaScript-objektin tai -taulukon nimi.

Kyselyiden tallentaminen välimuistiin

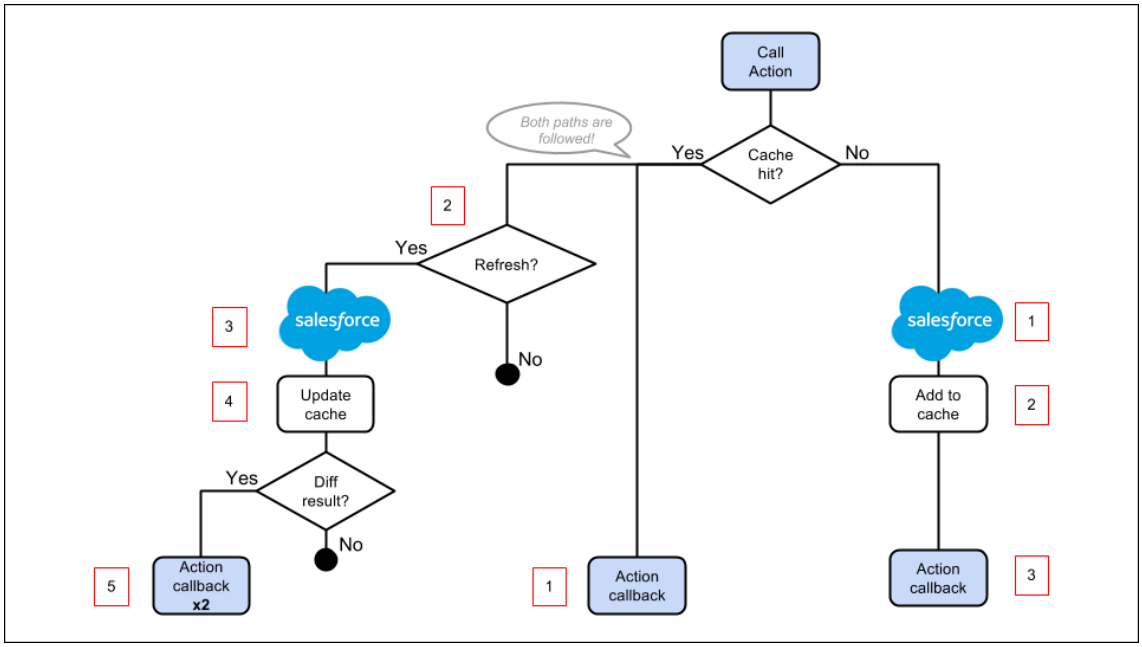
Mikäli komponentin controllerissa kutsuttava Apex-metodi ainoastaan lukee dataa eikä päivitä sitä, voidaan Apex-kutsuja tallentaa selaimen välimuistiin suorituskyvyn

parantamiseksi. Tästä on hyötyä varsinkin käyttäjille, joiden käyttämä internetyhteys on hidas tai epävakaa, sillä välimuistiin tallennettuja kyselyitä ei lähetetä palvelinpuolelle lainkaan.

Mikäli palvelinpuolen palauttama kysely halutaan tallentaa selaimen välimuistiin, voidaan tämä tehdä yksinkertaisesti käyttämällä syntaksia ”action.setStorable()”, jossa ”action” on komponentin controllerissa rakenteilla oleva Apex-kutsu jota ei vielä ole lähetetty palvelinpuolelle.

”setStorable()”-metodille voidaan antaa parametrinä ”ignoreExisiting”-niminen Boolean-attribuutti, joka arvolla ”True” ohittaa välimuistissa olevan arvon, jos ohjelmoija tietää palvelinpuolen datan päivittyneen esimerkiksi toisen komponentin Apex-kutsun vuoksi.

Kuvassa 9 nähdään, miten välimuistissa olevien tapahtumien aikajana kulkee.



Kuva 9. Välimuistissa olevien tapahtumien aikajana [13].

Jos kysely ei ole välimuistissa eli se ei vastaa yhtään välimuistissa olevaa tietuetta, toimitaan seuraavasti:

1. Pyyntö lähetetään suoraan palvelinpuolen Apex-controller-luokalle.

2. Mikäli palautunut kutsu onnistuu, se lisätään välimuistiin.

3. Mahdollinen Callback-funktio suoritetaan.

Mikäli kysely on jo välimuistissa, toimitaan seuraavasti:

1. Callback-funktio suoritetaan välimuistissa olevalla palautuksen arvolla.

2. Jos välimuistin arvo on vanhempi kuin sallittu päivitysaika, päivitetään välimuistissa oleva arvo. Päivitysaika on valmiiksi konfiguroitu Lightning-käyttöliittymälle ja Salesforce1-mobiilisovelluksille.

3. Pyyntö lähetetään palvelinpuolen Apex-controller-luokalle.

4. Mikäli palautunut kutsu onnistuu, se lisätään välimuistiin.

5. Jos päivitetty arvo on erilainen kuin välimuistissa oleva arvo, kutsutaan palvelinpuolen controller-luokkaa toistamiseen.

4.6 Palvelinpuolen Apex-controller

Pelkät Lightning-komponentit itsessään eivät yleensä riitä kokonaisen sovelluksen rakentamiseen, sillä sovellusta ladattaessa ja komponentteja piirrettäessä täytyy komponenttien käyttämä data saada haettua ja annettua niiden käytettäväksi. Tämä tapahtuu pavelinkutsulla Apexin puoleen. Jotta Apex-luokkaa voidaan komponentin controller-luokasta kutsua, tulee komponentille asettaa myös erillinen Apex-controller-luokka. Tämä tapahtuu komponentissa syntaksilla controller=”ApexController”, jossa ApexController on Apex-luokan nimi.

Jotta kutsuttavan Apex-luokan metodi tulee näkyville sitä kutsuvalle Lightning-controllerille, tulee kyseisen Apex-metodin olla luokkametodi. Näiden lisäksi koko metodin eteen on tultava Salesforcen oma ”@AuraEnabled”-annotaatio (esimerkkikoodi 16).

```
@AuraEnabled
public static List<Account> getAccountsByName(String searchKeyword)
{
      List<Account> queriedAccounts = new List<Account>();
      searchKeyword = '%' + searchKeyword + '%';

      queriedAccounts = [SELECT Id, Name, AccountNumber
                           FROM Account
                           WHERE Name LIKE :searchKeyword
                           LIMIT 50000];

      return queriedAccounts;
}
```

Esimerkkikoodi 16. Esimerkki Apex-controllerista, jota voidaan kutsua Lighting-controllerista.

Kaikki metodit eivät välttämättä ota argumentteja vastaan, mutta mikäli kutsuttu metodi tarvitsee argumentteja mukaansa, niitä voidaan asettaa ennen Apex-kutsun lähettämistä.

4.7 Tapahtumat ja komponenttien kommunikointi

Pääasiallinen tapa saada komponentit keskustelemaan keskenään on Lightning Eventeillä eli tapahtumilla. Tapahtumat ovat ainoa tapa saada tieto komponentista toiseen, mikäli nämä komponentit eivät ole sisäkkäin eivätkä näin tiedä toistensa olemassa olosta mitään. Komponentit rekisteröidään lähettämään tietynnimiset tapahtumat käyttäen ”aura:registerEvent”-komponenttia, ja nämä tapahtumat kootaan komponenttien controller-luokissa. Tapahtumille voidaan lisätä parametreja datan kuljettamiseksi komponentista toiseen. Kun tapahtuma on luotu ja siihen liitetty mahdollinen data, se tulee laukaista alustan käsiteltäväksi. Mikäli jokin komponentti on käsketty käsittelemään kyseisen koottu tapahtuma, ajetaan tämän komponentin controller-luokassa määrätty pätkä koodia. Käsiteltävälle komponentille on lisättävä komponentti ”aura:handler”.

Lightning-tapahtumia on kahta eri tyyppiä: Application ja Component. Ne eroavat siten, että Application-tapahtumat kuulutetaan kaikille tällä hetkellä olemassa oleville komponenteille ja ne kaikki voivat käsitellä tapahtuman, jos se on niille käsketty. Component-tapahtumat lähtevät hierarkkisesti ylöspäin ja vain komponentti, johon liitetty alemman tason komponentti laukaisee tapahtuman, pystyy käsittelemään kyseisen tapahtuman. Tällä voidaan rajoittaa sitä, etteivät kaikki komponentit käsittele turhaan tapahtumia, jotka eivät niille kuulu.

Tapahtumien hienous piilee siinä, että tarkoituksesta riippuen komponentit voivat toimia hyvin tietämättään tapahtumista yhtään mitään, mutta tehdä jotain tärkeää, kuten esimerkiksi päivittää dataa heti tapahtuman käsittelyn alkaessa.

Toinen mahdollinen tapa toteuttaa komponenttien välistä interaktiivisuutta on käyttää hyväksi attribuutteja ja ”aura:If”-komponentteja. Komponentit reagoivat reaaliajassa, mikäli niiden käyttämä attribuutti muuttuu. Tämän avulla sisäkkäin olevat komponentit voivat kommunikoida keskenään tarvitsematta mitään erillisiä tapahtumia. Esimerkkinä voidaan käyttää vaikkapa Modal-komponenttia, joka lisätään sivulle ja koodataan reagoimaan sivulla olevaan boolean-attribuuttiin. Kun sivun controller-luokka kääntää boolean-attribuutin true-arvoon, reagoi sivun Modal-komponentti siihen heti itsekseen. Insinöörityössä koodattiin tällainen Modal-komponentti, joka toimi käyttämättä Lightning-tapahtumia.

5 Modal-komponentti ja AccountSearch-sovelluksen rakentaminen

Insinöörityössä rakennettiin Salesforcen Lightning-sovelluskehyksellä Modal-komponentti, joka toimii täyden näytön ponnahdusikkunan tavoin. Modal-komponettia Salesforce ei suoraan tarjoa valmiina komponenttina, joten sen toteutus on tehtävä itse.

Toisena esimerkkinä rakennettiin Account-objektista tietoa hakeva ohjelma. Salesforce sisältää valmiin hakutoiminnon Account-objekteille, mutta työssä rakennettiin tämä objekteja hakeva pieni sovellus itse, sillä näin saatiin hyvä pintaraapaisu Lightning-kehityksen eri osa-alueisiin, kuten esimerkiksi komponenttien luontiin, Apex-controllerin kutsumiseen ja tapahtumien lähettämiseen ja käsittelyyn.

5.1 Modal-komponentti

Modal on yksinkertainen täyden näytön ponnahdusikkuna, jonka tarkoituksena on kohdistaa käyttäjän mielenkiinto pieneen osaan näytöstä. Modal estää pääikkunan käyttämisen esimerkiksi sumentamalla sen mutta pitää sen kuitenkin taustalla näkyvissä, jotta käyttäjä ei luule päätyneensä toiseen osaan sovellusta. Salesforce ei tätä työtä tehdessä tarjonnut omassa komponenttikirjastossaan Modal-komponenttia.

Modal-komponentti koostuu yhdestä komponentista, ja sitä ohjataan Modal-komponentin sisältävän komponentin controller-luokasta. Modal-komponentin toiminta perustuu sitä ohjaavaan ”showModal”-nimiseen boolean-attribuutiin, jota ”Aura:If”-komponentti seuraa. Mikäli ”showModal”-attribuutti näyttää true-arvoa, näytetään koko komponentti, joka sisältää tyylityksen taustan sumentamiseen. Itse komponentin koodi näkyy koodiesimerkissä 17.

```
<!-- Modal komponentti -->
<aura:component>

    <aura:attribute name="showModal" type="Boolean" default="false"/>

    <aura:attribute name="headerText" type="String" default="Modal"/>
    <aura:attribute name="footerButtons" type="Aura.Component[]"/>

    <aura:if isTrue="{!v.showModal}">
        <div class="slds-modal slds-fade-in-open">
            <div class="slds-modal__container">
                <div class="slds-modal__header">
                    <h2 class="slds-text-heading_medium">{!v.headerText}</h2>
                </div>
                <div class="slds-modal__content slds-p-around_medium">
                    {!v.body}
                </div>
                <div class="slds-modal__footer">
                    {!v.footerButtons}
                </div>
            </div>
        </div>
        <div class="slds-backdrop slds-backdrop_open"></div>
    </aura:if>

</aura:component>
```

Esimerkkikoodi 17. Modal-komponentin koodi.

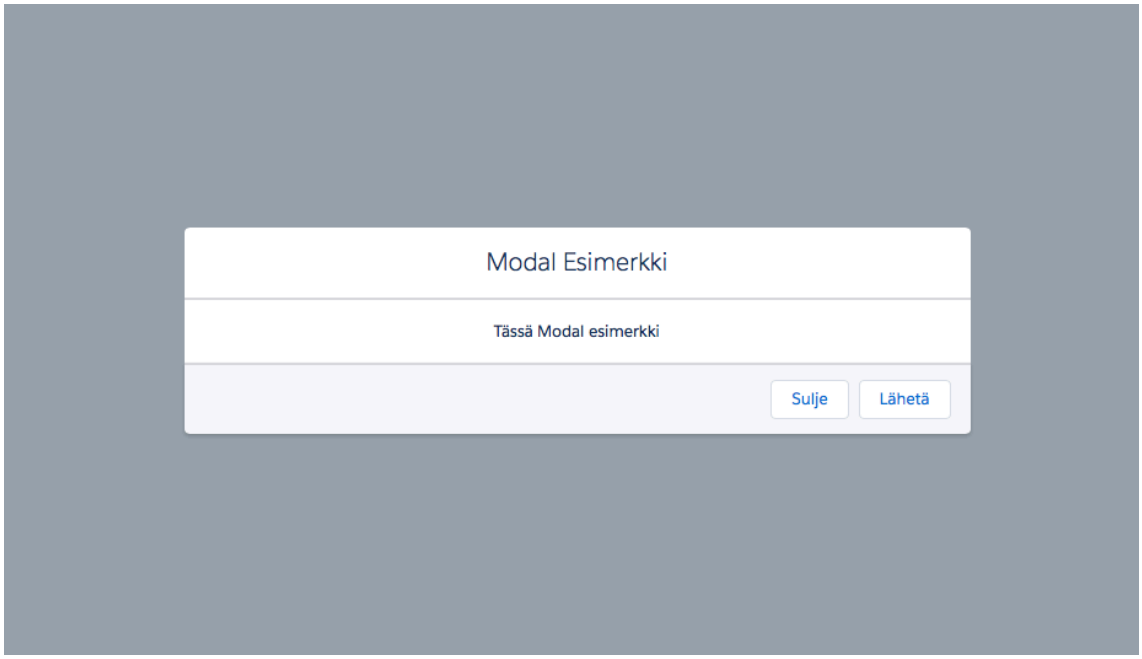
Koko komponentti attribuutteja lukuun ottamatta on Aura:If-komponentin sisällä, joka seuraa ”showModal”-attribuuttia. Jos sivu, jonne Modal-komponentti on asetettu, muuttaa tätä attribuuttia, reagoi Modal-komponentti muutokseen reaaliajassa joko näyttäen tai piilottaen itsensä. Tyylitysluokkina toimivat SLDS-tyylitykset, joissa on valmiina Modal-komponenttiin tarvittavat taustan sumennukset. Modal-komponentti ottaa vastaan lisäksi vapaaehtoisen attribuutin headerText, joka toimii ikkunan auetessa otsikkotekstinä. Esimerkkikoodin 17 rivillä 16 näkyy, mihin komponenttia luotaessa sen sisälle kirjoitetut palikat sijoitetaan, kun Modal-ikkuna näytetään. Lisäksi Modal-komponentti ottaa attribuuttina listan komponentteja. Sovelluksessa, jossa komponenttia ollaan luomassa (esimerkkikoodi 18), voidaan määrittää ”Set”-komponentin avulla lista nappuloista, jotka Modal-komponentin tulee näyttää omassa alapalkissaan.

```
<aura:application extends="force:slds">

     <aura:attribute name="showModal" type="Boolean" default="false" />
     <lightning:button label="Modal avaus" onclick="{!c.openModal} />

     <!-- Modal komponentti -->
     <c:modal headerText="Modal esimerkki" showModal="{!showModal}">

          <!-- Kaikki modalin sisälle tulevat komponentit tähän -->
          <div class="slds-align_absolute-center">Tässä Modal esimerkki</div>

          <!-- Modal komponentin alapalkin nappulat -->
          <aura:set attribute="footerButtons">

               <lightning:button label="Sulje" onclick="{!c.closeModal}"/>
               <lightning:button label="Lähetä" onclick="{!c.submitData}"/>

          </aura:set>
     </c:modal>

</aura:application>
```

Esimerkkikoodi 18. Koodiesimerkki Lightning-sovelluksesta siitä, miten Modal-komponentti luodaan.

Modal-komponentin sisältävällä sovelluksella tai komponentilla on oltava oma ”showModal”-attribuutti, jota ohjataan sovelluksen tai komponentin omassa controller-luokassa. Kaikki esimerkkikoodin 18 rivien 8 ja 17 välille kirjoitetun koodin tulkitaan olevan Modalin ”body”-tagien sisällä ja sijoitetaan Modal-komponentissa (esimerkkikoodi 17) riville 16.

Metodit jotka ohjaavat Modalin näkyvyyttä, kirjoitetaan komponentille, johon Modal komponentti upotetaan. Niiden ainoa tehtävä on muuttaa Modal-komponentin seuraaman attribuutin arvoa (esimerkkikoodissa 18 tämä on rivin 3 ”showModal”-attribuutti). Näin päästään lisäksi käsittelemään Modalia toteuttavan komponentin dataa itse Modalissa, koska mahdolliset käsiteltävät komponentit kirjoitetaan suoraan Modal-komponentin sisään. Modal-komponentin ei näin tehtynä tarvitse tietää itse toteutettavasta logiikasta mitään, vaan sen ainoa tehtävä on näyttää ja piilottaa toteutettu logiikka haluttuna aikana. Kuvassa 10 näkyy avoimena oleva Modal-komponentti.



Kuva 10. Avoin Modal-komponentti.

5.2 AccountSearch-sovellus

Seuraavaksi työssä rakennettiin sovellus, jonka avulla käyttäjä pystyy etsimään Salesforcessa olevia Account-objekteja. Ohjelma sisältää nimisuodatusmahdollisuuden ja näyttää valitun objektin tiedot etsintäpalkin vieressä. Ohjelma hakee dataa sitä mukaa, kuin käyttäjä kirjoittaa lisää.

Ohjelma tuli koostumaan yhdestä Lightning-sovelluksesta, johon kaikki komponentin sijoitetaan, kolmesta eri Lightning-komponentista (AccountDetails, AccountSearchBar ja AccountList), kahdesta Lightning-tapahtumasta ja yhdestä Apex-controller-luokasta.

AccountSearchBar ja AccountSearchEvent

Esimerkkikoodissa 19 nähdään, kuinka AccountSearchBar-komponentti sisältää syöttömahdollisuuden halutulle hakusanalle, ja sille annetaan mahdollisuus laukaista AccountSearchEvent-nimisiä tapahtumia. Kun hakusanaa kirjoitetaan, kutsutaan komponentin controller-luokkaa aina hakusanan muutoksen tapahtuessa.

```
<!-- AccountSearchBar komponentti -->
<aura:component >
    <aura:registerEvent name="SearchEvent" type="c:AccountSearchEvent"/>

    <lightning:input type="search"
                     label="Etsi"
                     name="search"
                     placeholder="Kirjoita hakusana tähän"
                     onchange="{!c.searchInputChanged}"/>

</aura:component>
```

Esimerkkikoodi 19. AccountSearchBar-komponentin toteutus.

Kutsuttava controller-metodi laukaisee AccountSearchEvent-tapahtuman joka kerta, kun hakusana muuttuu. Tapahtumalle annetaan parametrinä tämän hetkinen kirjoitettu hakusana. AccountSearchBar komponentin controller-luokka on esimerkkikoodissa 20.

```
({
     searchInputChanged : function(component, event, helper)
     {
          var searchEvent = $A.get("e.c:AccountSearchEvent");
          searchEvent.setParams(
          {
               "keyword": event.getSource().get("v.value")
          });
          searchEvent.fire();
     }
})
```

Esimerkkikoodi 20. AccountSearchBar-komponentin controller-luokka.

AccountList

AccountList-komponentin tarkoituksena on näyttää listaa löydetyistä asiakkaista. Esimerkkikoodissa 21 nähdään, kuinka komponentilla kutsutaan alustusmetodia rivillä 4. Alustusmetodissa haetaan tietokannasta kaikki mahdolliset asiakkaat, jotta lista saadaan alustettua. Rivillä 8 käydään tämä lista lävitse ja näytetään kaikki palautetut asiakkuudet.

```
<!-- AccountList komponentti -->
<aura:component controller="AccountSearchController">
    <aura:attribute name="accounts" type="Account[]" default="[]"/>

    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
    <aura:handler event="c:AccountSearchEvent"
                  action="{!c.handleSearchEvent}"/>

    <ul class="slds-list--vertical slds-has-cards
                slds-has-block-links--space
                slds-has-list-interactions">
        <aura:iteration items="{!v.accounts}" var="account">
            <li class="slds-list__item
                       slds-p-around_medium
                       slds-m-bottom_small">

                <div id="{!account.Id}"
                     class="slds-p-around_medium"
                     onclick="{!c.accountSelected}">

                    <div class="slds-p-horizontal_x-small">
                        {!account.Name}
                    </div>
                    <div class="slds-p-horizontal_x-small">
                        {!account.AccountNumber}
                    </div>

                </div>

            </li>
        </aura:iteration>
    </ul>
</aura:component>
```

Esimerkkikoodi 21. Accountlist-komponentin koodi.

AccountList-komponentin tarkoituksena on myös hallita AccountSearchBar-komponentin lähettämää AccountSearchEventiä. Kun tapahtuma laukaistaan, käsittelee AccountList-komponentti sen käyttäen metodia ”handleSearchEvent” (esimerkkikoodissa 22). Tämä metodi kyselee komponentin Apex-controller-luokalta niitä asiakkuuksien nimiä, jotka osuvat tapahtumasta saatuun hakusanaan. Kun uusi rajattu hakutulos palautuu, se laitetaan saman ”accounts”-attribuutin tilalle, jolloin iteration-komponentti reagoi muuttuneeseen attribuuttiin piirtämällä listan uudelleen.

```
({
	doInit : function(component, event, helper)
	{
		var getAllAccountsAction = component.get('c.getAllAccounts');
		getAllAccountsAction.setCallback(this, function(response)
		{
			if(response.getState() === 'SUCCESS' && component.isValid())
			{
				component.set('v.accounts', response.getReturnValue());
			}
		});
		$A.enqueueAction(getAllAccountsAction);
	},

	handleSearchEvent : function(component, event, helper)
	{
		var searchAccountsAction = component.get('c.getAccountsByName');
		searchAccountsAction.setParams(
		{
			"searchKeyword" : event.getParam("keyword")
		});
		searchAccountsAction.setCallback(this, function(response)
		{
			if(response.getState() === 'SUCCESS' && component.isValid())
			{
				component.set('v.accounts', response.getReturnValue());
			}
		});
		$A.enqueueAction(searchAccountsAction);
	},

	accountSelected : function(component, event, helper)
	{
		var selectedId = event.currentTarget.id;
		var selectedEvent = $A.get("e.c:AccountSelectedEvent");
		selectedEvent.setParams({ "selectedAccountId": selectedId });
		selectedEvent.fire();
	}
})
```

Esimerkkikoodi 22. AccountList-komponentin controller-luokka.

Apex-controller-luokassa palautetaan jokaiselta asiakkuudelta kolme kenttätietoa: Id, Name ja AccountNumber. Nämä tiedot ovat Salesforcen tietokannassa. Kun komponentissa iteroidaan palautettua listaa lävitse, näytetään kaikista listan tietueista kenttä arvot Name ja AccountNumber ja niistä tehdään listan tietue.

Mikäli käyttäjä klikkaa yhtä listan tietueista, kutsutaan controller-luokan metodia ”accountSelected”, joka saa funktiolle annetun ”event”-nimisen parametrin tiedoista sen listan arvon, jota käyttäjä on klikannut. Tämän jälkeen funktio lähettää AccountSelectedEvent-tapahtuman liikkeelle, jonka parametreiksi annetaan valitun asiakkuuden Id-tieto. Tunnistamisessa käytetään Id-kenttää pelkän nimen sijasta, sillä Id-kentän arvo

generoidaan Salesforcessa automaattisesti, kun asiakkuus luodaan tietokantaan, eikä se voi olla sama kuin jollain muulla objektilla tai tietueella.

AccountDetails

AccountDetails komponentin (esimerkkikoodi 23) tehtävä on näyttää käyttäjälle enemmän tietoa sovelluksessa valitusta asiakkuudesta. Kun AccountList-komponentissa valitaan jokin hakutuloksen asiakkuus, lähetetään AccountList-komponentin controller-luokassa (esimerkkikoodi 22) AccountSelectedEvent, joka sisältää parametrinä valitun asiakkuuden Id-tiedon.

```
<!-- AccountDetails komponentti -->
<aura:component controller="AccountSearchController">

    <aura:attribute name="selectedAccount" type="Account" default="{}"/>
    <aura:handler event="c:AccountSelectedEvent"action="{!c.changeAccount}"/>

    <div class="details">
        <h1>{!v.selectedAccount.Name}</h1>
        <h3>{!v.selectedAccount.AccountNumber}</h3>
        <h3>{!v.selectedAccount.Type}</h3>
        <br/>
        <p>{!v.selectedAccount.BillingAddress.street}</p>
        <p>{!v.selectedAccount.BillingAddress.city}</p>
    </div>

</aura:component>
```

Esimerkkikoodi 23. AccountDetails-komponentti. Tällä komponentilla on sama Apex-controller-luokka kuin AccountList-komponentilla.

Esimerkkikoodissa 24 nähdään, miten AccountDetails-komponentti käsittelee tämän AccountSelectedEvent-tapahtuman ja kyselee Apex-tietokannalta tiedot asiakkuudesta sillä Id-arvolla, joka oli tapahtumassa parametrinä. AccountDetails-komponentin käyttämä Apex-metodi palauttaa asiakasobjektista enemmän kenttäarvoja, jolloin komponentissa voidaan viitata useampaan objektin arvoon kuin AccountList-komponentissa, jonka kutsuttavana oleva Apex-metodi palauttaa vain muutaman kenttäarvon.

```
({
    changeAccount : function(component, event, helper) {
        var selectedAccountId = event.getParam("selectedAccountId");
        var action = component.get("c.getAccountById");
        action.setParams(
        {
            "accountId": selectedAccountId
        });
        action.setCallback(this, function(response)
        {
            if(response.getState() === 'SUCCESS' && component.isValid())
            {
                component.set("v.selectedAccount", response.getReturnValue());
            }
        });
        $A.enqueueAction(action);
    }
})
```

Esimerkkikoodi 24. AccountDetails-komponentin controller-luokka.

5.2.1 Apex controller ja valmis sovellus

Tarvittu Apex-controller-luokka sisältää yhteensä kolme metodia (esimerkkikoodi 25) ja mikäli Apex-controller-luokka olisi ollut monimutkaisempi tai jos sovelluksen komponenteista haluttaisiin tehdä geneerisempiä, niin että niitä pystyisi käyttämään muissakin sovelluksissa, olisi Apex-luokka kannattanut jakaa useaan luokaan niin, että kaksi komponenttia eivät olisi riippuvaisia samasta luokasta. Päätin kirjoittaa metodit yhteen luokkaan sen takia, että kyseessä olevat komponentit eivät ole kovin geneerisiä eli niitä käytetään vain tämän sovelluksen rakentamisessa.

```
public class AccountSearchController
{
    @AuraEnabled
     public static List<Account> getAllAccounts()
     {
       List<Account> allAccounts = new List<Account>();
       allAccounts = [SELECT Id, Name, AccountNumber
                          FROM Account
                          LIMIT 50000];
       return allAccounts;
     }

     @AuraEnabled
     public static Account getAccountById(Id accountId)
     {
       Account matchedAccount = new Account();
       matchedAccount = [SELECT Id, Name, AccountNumber, Type, BillingAddress
                          FROM Account
                          WHERE Id = :accountId];
          return matchedAccount;
     }

     @AuraEnabled
     public static List<Account> getAccountsByName(String searchKeyword)
     {
       List<Account> queriedAccounts = new List<Account>();

       // "%" syntaksilla annetaan hakusanalle wildcard arvo.
       searchKeyword = '%' + searchKeyword + '%';

       queriedAccounts = [SELECT Id, Name, AccountNumber
                          FROM Account
                          WHERE Name LIKE :searchKeyword LIMIT 50000];

       return queriedAccounts;
     }
}
```

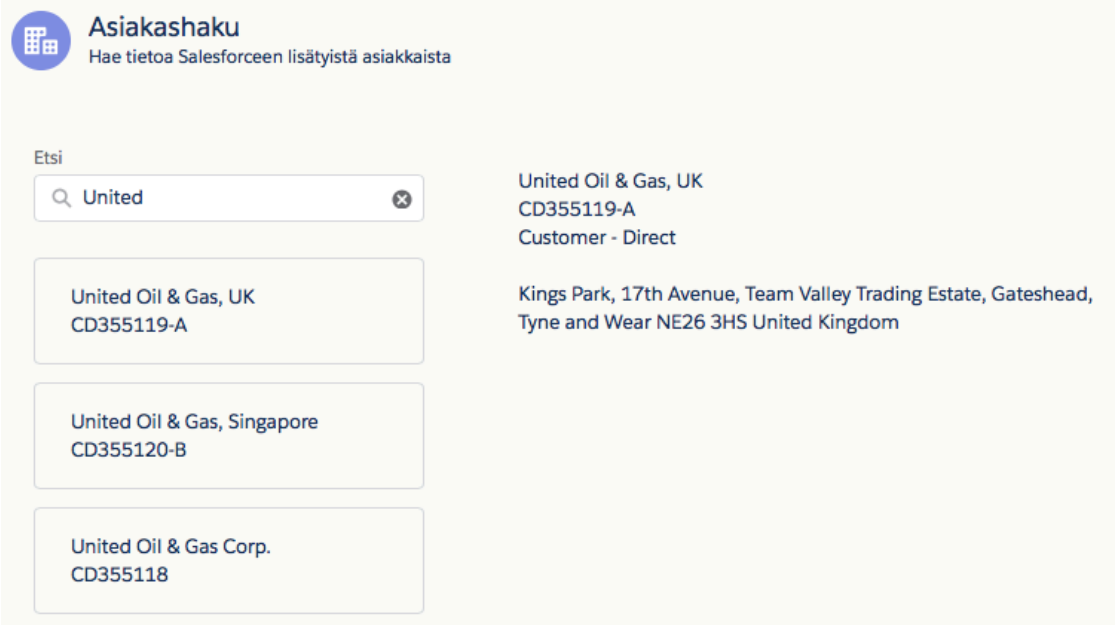
Esimerkkikoodi 25. AccountSearchController Apex -luokka.

Lopuksi kaikki esitellyt komponentit tulee vielä laittaa omaan sovellukseensa (esimerkkikoodi 26). Itse rakennettujen komponenttien sijainnit näkyvät riveillä 24, 26 ja 31. Rivin 6 Icon-komponentti tarjoaa Salesforcen omia kuvaketiedostoja käytettäväksi ilman erillisiä resurssilatauksia Salesforcen alustalle.

```
<aura:application extends="force:slds">
    <div class="slds-page-header slds-p-around_medium">
        <div class="slds-media">

            <div class="slds-media__figure">
                <lightning:icon iconName="action:new_account" size="small"/>
            </div>

            <div class="slds-media__body">
                <p class="slds-page-header__title
                          slds-truncate slds-align-middle">
                    Asiakashaku
                </p>
                <p class="slds-text-body--small slds-page-header__info">
                    Hae tietoa Salesforceen lisätyistä asiakkaista
                </p>
            </div>

        </div>
    </div>

    <div class="slds-grid slds-wrap">

        <div class="slds-col slds-size_5-of-12 slds-p-around_x-large">
            <div class="slds-p-bottom_large">
                <c:AccountSearchBar />
            </div>
            <c:AccountList />
        </div>

        <div class="slds-col slds-size_7-of-12 slds-p-around_x-large">
            <div class="slds-p-top_medium">
                <c:AccountDetails />
            </div>
        </div>

    </div>
</aura:application>
```

Esimerkkikoodi 26. Sovelluksen rakenne. Rakenteessa näkyy komponenttien sijoittelu ja niiden luonti.

Lopputuloksena (kuva 11) saatiin sovellus, joka hakee reaaliajassa hakusanan perusteella asiakkuusdataa. Haussa käytetty objekti on nimeltään Account, mutta Apex-luokkaa muokkaamalla voidaan sovellus helposti muokata hakemaan dataa muista kuin Account-objekteista.



Kuva 11. Lopputuloksena saatu AccountSearch-sovellus.

6 Yhteenveto

Insinöörityössä perehdytettiin kehittäjä Salesforcen Lightning-sovelluskehykseen ja sen komponenttipohjaiseen ajattelutyyliin ja toteuttamiseen.

Työssä perehdyttiin Lighting-komponenttien rakenteeseen, syntaksiin ja controller-luokkiin, ja siinä käytiin läpi myös tärkeä komponenttien välinen kommunikointi käyttäen Lightning-tapahtumia. Työssä tehtiin esimerkkikomponentti, jollaista Salesforce ei työtä tehdessä tarjonnut suoraan standardikomponenttina, ja Lighting-sovellus, jolla haetaan ja näytetään tietoa Salesforce-ympäristön tietokannasta. Tämä komponentti ja sovellus näyttävät sekä komponenttien välistä kommunikointia että tiedon lähettämistä ja hakemista tietokannasta.

Insinöörityöraportin on tarkoitus auttaa ohjelmistokehittäjiä aloittamaan Lightning-sovelluskehyksellä komponenttien kehittäminen. Työ oli kaiken kaikkiaan onnistunut ja antaa lukijalle hyvät perustiedot Lightning-sovelluskehyksestä. Lightning-sovelluskehys ja Salesforce-alusta ovat hyvin laajoja aiheita, ja tätä opinnäytetyötä voi laajentaa vielä syvemmälle Lightning-sovelluskehyksen ominaisuuksiin, kuten esimerkiksi sovellusten näyttämiseen Salesforce1-alustalla, palvelinpuolelta lähetettävien platform-tapahtumien käsittelyyn ja luontiin tai Lightning-komponentteihin Visualforce-sivulla.

Lähteet

1 McCormic, Moira. 2016. 7 Benefitsf Software-as-a-Service (SaaS). Verkkoaineisto. BlackCurve. <https://blog.blackcurve.com/7-benefits-of-software-as-a-service-saas>. Luettu 2.3.2018.

2 Markovski, Miso. 2017. Top 10 CRM Software Vendors and Market Forecast 2016-2021. Verkkoaineisto. Apps Run The World. <https://www.appsruntheworld.com/top-10-crm-software-vendors-and-market-forecast/>. Luettu 2.3.2018.

3 Aalto, Peter. 2016. Asiakkuudenhallintajärjestelmän integrointi ajanvarausjärjestelmään. Insinöörityö. Metropolia Ammattikorkeakoulu. Theseus-tietokanta.

4 Ravattinen, Tero. 2016. Salesforcen ylläpitäminen. Insinöörityö. Metropolia Ammattikorkeakoulu. Theseus-tietokanta.

5 Forcedotcom / Aura. Verkkoaineisto. Github. <https://github.com/forcedotcom/aura >. Luettu 4.3.2018.

6 Rosenbaum, Mike. 2014. Introducing Salesforce1 Lightning. Verkkoaineisto. Salesforce. <https://www.salesforce.com/blog/2014/10/introducing-salesforce1-lightning.html/>. Luettu 4.3.2018.

7 Bootstrap. Verkkoaineisto. Bootstrap. <https://getbootstrap.com/>. Luettu 6.4.2018.

8 Cummins, Stephen. 2016. Demystifying the Numbers behind Salesforce.com's AppExchange. Verkkoaineisto. Medium. <https://medium.com/understanding-as-a-service-uaas/demystifying-the-numbers-behind-salesforce-com-s-appexchange-60b3cedbc01f>. Luettu 7.4.2018.

9 AppExchange, The Salesforce Store. Verkkoaineisto. Salesforce. <https://appexchange.salesforce.com/>. Luettu 8.4.2018.

10 Share Lightning Out Apps with Non-Authenticated Users. 2016. Verkkoaineisto. Salesforce. <https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/lightning_out_public_apps.htm>. Luettu 9.4.2018

11 Topalovich, Mike. 2016. Why Does it Matter if I Use a Space Between .THIS and a CSS Class Name in the Lightning Component Bundle Style Resource. Verkkoaineisto. Topalovich. <http://www.topalovich.com/2016/12/why-does-it-matter-if-i-use-a-space-between-this-and-a-css-class-name-in-the-lightning-component-bundle-style-resource/>. Luettu 12.4.2018.

12 Lightning Components Developer Guide, Component Reference. Verkkoaineisto. Salesforce. < https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/aura_compref.htm>. Luettu 12.4.2018.

13 Lightning Components Developer Guide, Lifecycle of Storable Actions. Verkkoaineisto. Salesforce. <https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/controllers_server_storable_lifecycle.htm>. Luettu 8.4.2018.